AALTO UNIVERSITY SCHOOL OF SCIENCE AND TECHNOLOGY

Faculty of Electronics, Communications and Automation

Department of Communications and Networking

Risto Loikala

A Bit Error Measurement Set-Up for a Radio Relay System and a Survey of Error Control Techniques

Thesis submitted for examination for the degree of Master of Science in Technology
Espoo 3.5.2010

Thesis supervisor:
Prof. Riku Jäntti
Thesis instructor:
Markku Liinaharja

AALTO UNIVERSITY SCHOOL OF SCIENCE AND TECHNOLOGY

ABSTRACT OF THE MASTER'S THESIS

Author: Risto Loikala

Title: A Bit Error Measurement Set-Up for a Radio Relay System and a Survey of Error Control Techniques

Date: 3.5.2010 Language: English Number of pages: 5 + 73

Faculty: Faculty of Electronics, Communications and Automation

Department: Department of Communications and Networking Code: S-72

Supervisor: Prof. Riku Jäntti

Instructor: Markku Liinaharja

The properties of radio channels vary greatly in the wireless medium and it is important to know how a signal behaves in such. A binary signal, depending on the path that it travels, can experience various changes in it as the surrounding environment affects it. This will results in errors in the data at the receiver that need to be avoided for a successful transmission of data to occur.

The goal of this thesis is to see how we could set up and measure the errors occurring in a radio relay system at the bit level, take a look at various manners of error control coding that are or have been in use and channel models that are used to model the conditions that a transmitted signal may face.

For the bit-level measurement of channels, a program was programmed that can send and receive data, which is known on the bit-level. The information that has passed through the channel can then be compared at the receiver to its original form to detect errors on the bit-level and this information can then be used to create a model of the error behaviour of the channel. This information can then be used for determining how certain error control coding would fare in the channel. Measurements were done using the program in laboratory conditions with an actual radio relay system. The measurements were done with an adjustable resistor placed at the end of a wired connection connecting the two relay systems, which could be adjusted to simulate differing distances between the two relays.

The measurements done gave us results that pointed towards a certain model and showed that the program can be used for its intended purpose.

Keywords: Error control coding , radio channel, measurement, channel model, programming

AALTO-YLIOPISTON TEKNILLINEN KORKEAKOULU

DIPLOMITYÖN TIIVISTELMÄ

Tekijä: Risto Loikala

Työn nimi: Bittivirheiden mittausjärjestelmä radiorelesysteemiä varten ja katsaus virheenkorjausmenetelmiin

Date: 3.5.2010 Kieli: Englanti Sivumäärä: 5 + 73

Tiedekunta: Elektroniikan, Tietoliikenteen ja Automaation Tiedekunta

Laitos: Tietoliikenne- ja tietoverkkotekniikan laitos Koodi: S-72

Valvoja: Prof. Riku Jäntti

Ohjaaja: Markku Liinaharja

Radiokanavien ominaisuudet vaihtelevat suuresti langattomassa viestinnässä ja on tärkeää tuntea kuinka signaali käyttäytyy siinä. Binäärinen signaali, riippuen polusta jonka se kulkee, voi kokea erilaisia muutoksia kun sitä ympäröivä ympäristö vaikuttaa siihen. Tämä johtaa virheisiin vastaanottimen saamassa tiedossa joita meidän tarvitsee välttää jotta onnistunut tiedon siirto voisi tapahtua.

Tämän diplomityön tarkoituksena on katsoa kuinka me voisimme mitata virheitä bitti-tasolla radiorelejärjestelmässä, tarkastella erilaisia virheenkorjauskoodeja joita on käytetty tai käytetään ja kanavamalleja joita käytetään kuvaamaan niitä olosuhteita joita lähetetty signaali voi kohdata.

Kanavien bitti-tason mittauksia varten kehitettiin ohjelma joka voi lähettää ja vastaanottaa dataa joka tunnetaan bitti-tasolla. Kun kyseinen tieto on matkustanut kanavan läpi, sitä voidaan verrata vastaanottimessa sen alkuperäiseen muotoon jonka perusteella pystymme havaitsemaan virheet bitti-tasolla ja tästä tiedosta pystymme luomaan mallin kanavan käyttäytymiselle. Siitä vastaavasti pystymme päättelemään miten jokin virheenkorjauskoodi toimisi kyseisessä kanavassa. Mittauksia suoritettiin käyttäen tätä ohjelmaa laboratorio-olosuhteissa oikealla radiorelejärjestelmällä. Nämä mittaukset suoritettiin yhdistämällä relejärjestelmät kaapelilla ja asettamalla säädettävä vastus toiseen päähän jolla pystyimme simuloimaan vaihtelevaa etäisyyttä järjestelmien välillä muuttamalla vastuksen kokoa.

Työssä tehdyt mittaukset antoivat meille tuloksia jotka osoittivat kohti erästä kanava mallia ja osoittivat että ohjelmaa voidaan käyttään sen tarkoitettuun käyttötarkoitukseen.

Avainsanat: Virheenkorjauskoodaus , radiokanava, mittaus, kanavamalli, ohjelmointi

Preface

This thesis was written concerning the work I did at the Department of Communications and Networking at the Helsinki University of Technology and I am grateful for the opportunity to work there and all the support offered to me by the people there.

I would like to thank Seppo Saastamoinen for approaching me about the possibility to work at the project and making it possible for me to have had this opportunity and Riku Jäntti for acting as my supervisor and my instructor Markku Liinaharja for his patience and help in writing this work.

Most of all, I would like to thank my sister Katja, because without her I would have never had the chance to study at the university.

Table of Contents

Chapter 1: Introduction

1.1 Scope and Purpose of the thesis

The main aim of this thesis is the overview and look at measuring radio channels on a bit-level through a computer program created specifically for this thesis as well as taking a look at radio communications and error control coding. Error control coding itself was one of the main reasons for the creation of the program and knowing how a channel behaves on a bit level as data travels through it gives useful information to make decisions on what type of error correction would be most effective at ensuring that data is received as it was sent.

We first go through an introductory section before discussing radio channels and some of the models used to describe them. This is followed by discussion of error correction methods, with an emphasis on Forward Error Correction and also Hybrid Automatic Repeat-reQuest. We then take a detailed look at how the measurement program functions and what practical decisions were made in its implementation. Actual measurements, the equipment used and the results gained from these measurements are discussed in the next chapter and finally, in Chapter 6, a short review of the thesis is given. The code for the transmitter side is included in Appendix 1 and the code of the receiver program in Appendix 2.

1.2 Digital Communications

The roots of digital communications reach even farther back in time than most people think, with Morse and his line telegraphy and Morse code from the 1840's being one of the first examples of a form of electrical digital communication. Despite the existence of digital communications for so long, it has taken until the recent decades that digital communications has overtaken its analog counterpart.

The basis of communications lies in the transportation of information from one point to another via some kind of a medium. With analog communications, the information is transported by a continuously varying signal that is used to modulate a carrier wave. In contrast, a digital signal is comprised of a signal with discrete levels that change after discrete time periods and is now the type used widely in Internet traffic and most consumer electronics.

1.3 Radio

The radio was originally developed by Thomas Edison and called "wireless telegraphy", before the term was changed to "radio" during the 20th century. It refers to the transmissions of signal through modulation of electromagnetic waves that then travel through the air. These signals are then picked up as the radio waves pass through an antenna, which works as a conductor, and induce a current into it. The original information signal can then be extracted from this current.

While commercial radios used to be mostly analogue, a lot of new systems nowadays are digital, allowing better transportation of data through them for various purposes. But with varying conditions, the need for efficient data transmission continues to grow and the need for the use of radio even in adverse conditions where errors in transmission can occur remains.

1.4 Channels and Error Correction

What ultimately determines the performance of a communications system besides the limits of the technology involved is the medium through which the data is transferred. This medium may be a copper cable, an optical fibre or a wireless link. These communication channels can generally be divided into two groups. In the case where a solid medium exists and connects the transmitter and receiver, the channel is called a wired channel. If this sort of solid link between the two is missing, the channel is called a wireless channel. Compared to wired channels, wireless channels are by nature more

complex, with the state of the channel possibly changing in very short spans of time and it is this behaviour that makes communications via wireless channels a comparatively more difficult task.

The varied nature of the environments that wireless channels are used in also creates different needs for the technology being used, especially in the case of mobile technology where the location of the transmitter and receiver may actually move from one kind of environment to another. And whether this environment is suburban, urban, indoor, forested or even orbital, communications must be successfully completed through it.

As there is ultimately always degradation in the transmission which causes the receiver to possibly gain erroneous data, there needs to be a way to recover from these errors. Error correction codes offer various ways to do this, ranging from very simple coding schemes that offer limited protection against errors to more complex ones that rely on several decoding iterations to recover even from very bad transmission conditions. The problem lies in knowing what kind of error correction scheme is the correct one. The various environments where wireless communications are used lead to different manners in which the data transmitted will deteriorate and experience errors. Also performance issues need to be addressed. How much of the transmission capacity can be dedicated to error correction? This tends to be more of an issue in real-time communications such as video conferences, rather than where pure data is being transmitted, such as a Word-document over the Internet.

To choose an appropriate error correction method, the behaviour of the channel needs to be known. Several models exist that describe the behaviours of various environments, which can be used to design or choose an error correction scheme, but being able to find out the exact behaviour of the channel in question is a desirable thing.

But how does one do this? There are several different ways, ranging from measuring the received power spectrum of a signal to see how it behaves in the environment [1] to observing lost packets. Overall, channel characterization can be done

basically with any transmitter-receiver configuration. But for deciding on an error correction method, the most useful information could be to know the bit-level error statistics.

For this purpose, a program which generates semi-random data which can be then recreated at the receiver was written. Observing and comparing the received data to the reference data, we can then find out how errors occurred in the data during the time it travelled through the channel and use this information in choosing or developing an error control scheme. Because the program was originally made with a particular radio relay system in mind, we will concentrate solely on error correction codes using hard decision decoding. The results of the measurements done by the raw data program will also be compared to ones done via the Iperf network testing tool to see if there are any commonalities between the results from the two tools.

Chapter 2: Radio Channels and Models

2.1 Radio Channel

When considering the propagation of a signal in a radio channel, the influences on the signal travelling through the channel can generally be divided into three different categories [2]. Each of these has a detrimental effect on the signal and can cause the receiver to interpret the received data incorrectly.

- Noise
- Interference
- Attenuation

2.1.1 Noise

Noise is always present in wireless communication systems and adds a distorting element to the signal. The sources for the noise are various, ranging from atmospheric disturbance to machinery and devices made by humans. The first is classified as thermal noise while the second is a human-made source. Thermal noise is caused by the thermal agitation of the charge carriers inside an electrical component and is unavoidable in those components.

This type of noise is called white noise as it is uniform in power over all frequencies, in the same way as white light contains all the frequencies of light. The general channel model which takes this type of noise into account is the Additive White Gaussian Noise (AWGN) channel. While it is a good model for satellite and space communications, as it can model things such as the noise radiating from the sun, it is not a good model for terrestrial communications, as it leaves out things such as interference, multipath propagation and terrain features. However, it is not a useless

model and is used to simulate background noise in combination with models which can simulate the behaviour of other parts of the channel that AWGN cannot.

Man-made noise is a more complicated topic. It is the unintended, radiated product of machines. While their main purpose is some other function, due to wiring or turning electronics on and off, they radiate electromagnetic energy which can disturb wireless communications. The good thing is that this noise in generally fairly low range[2] and should not generally be a problem for radio communications, unless either the receiver or transmitter has been placed near other radiating equipment. This is also a good thing because to model this type of noise, many parameters have to be considered and they are always specific to that particular case or environment. These include average total power, power spectrum, pulse heights and rates and others. But in the end, these noise sources would only need to be considered within an urban environment or within buildings as a possible source of errors in the reception.

2.1.2 Interference

Interference is similar to noise in that it's a distorting element, but the difference between the two lies in the purpose of the sources. While the noise sources task is not the production of the disturbance, the job of the source of the interference is to produce that signal. This does not necessary mean that the purpose of the signal is to actually interfere with other signals. If the interference is intentional, the term "jamming" is generally used instead of interference. In the case of jamming, the interfering signal is used to overpower and cover the actual desired signal and to keep the receiver from being able to recognize the signal at all. As such, error correction codes will not solve this problem and we will concentrate on non-jamming interference.

In general interference itself can be divided into co-channel and adjacent channel interference[2]. In co-channel interference, a lone transmitter interferes with the operations of another transmitter who is using the same radio frequency at the same time. The receiver, which is receiving the wanted signal from one of the transmitters,

also happens to receive the generally weak signal from the second transmitter, which can then cause the receiver to misinterpret the received data.

A good example is a cellular system where the cells operate on the same frequency band. The cells all have a radius of R and the separation between the cells is D. If we assume that the receiver is at the edge of the cell, then the distance between the transmitter and the receiver is R. Denoting the overall power of the transmission by P_0, we can then denote the power of the received signal as P_r , where P_r after path loss can be calculated as shown in Equation 2.1, where k is dependent on the environment in question.

$$P_r = \frac{P_0}{R^k} \tag{2.1}$$

Following from this, we can then describe the power from an interfering transmitter at the receiver by P_i, which is given by 2.2.

$$P_i = \frac{P_0}{(D-R)^k} \tag{2.2}$$

Because of the structure of a cellular system, consider then that there exist n equally strong interfering transmitters in the neighbouring cells, which would be the worst case scenario. In this case, we find out that the ratio between the power from the intended transmitter and the interfering transmitters, which is known as the carrier-to-interference ratio, with 2.3.

$$CIR = \frac{P_r}{n \cdot P_i} = \frac{1}{n} \cdot (\frac{D-R}{R})^k \tag{2.3}$$

What we can see from this is that increasing the transmission powers of the transmitters doesn't help with the ratio at all, therefore proving no improvement for

the system. As the power from the interfering sources can be considered noise, the CIR can be compared to signal-to-noise ratio.

With adjacent channel interference, a signal transmitted on a different but nearby frequency can produce interference in the receiver. This is caused mainly by imperfect filtering at the receiver, which allows some of the signal from an adjacent channel to leak over, but can also be caused by poor frequency control in either the reference or the interfering channel.

2.1.3 Attenuation

As we were able to divide the influences on signal propagation into three different parts, attenuation itself can also be divided into three categories[2].

- Path Loss
- Shadowing
- Fading

In the measurements which are discussed and described later, we are mainly concentrated on attenuation and specifically path loss. Path loss is the phenomenon where the transmitted signal experiences a loss in power, which depends on the square of the distance of the transmission and the square of the frequency. In Maxwell equations, the path loss in free-space propagation where a line of sight is assumed between the transmitter and receiver is given by

$$\frac{P_0}{P_t} = \left(\frac{\lambda}{4\pi d}\right)^2 G_{Tx} G_{Rx} \qquad (2.4)$$

where P_0 is the received power, P_t is the transmitted power, λ is the wavelength, d is the distance the signal has travelled in meters, G_{Tx} is the gain of the transmitter antenna and G_{Rx} is the gain of the receiver antenna. The same equation can also be expressed in dB as

$$\frac{P_0}{P_t}(dB) = 20\log\left(\frac{\lambda}{4\pi d}\right) + 10\log(G_{Tx}) + 10\log(G_{Rx}) \qquad (2.5)$$

Shadowing is non-deterministic and causes fluctuations in the received signal strength at points with the same distance to the transmitter. It is composed of the effects on the signal by several phenomena that can affect a transmitted signal, such as reflections from buildings, diffraction, refraction, scattering and absorption. However, the average over several received signals for the same distance yields the same value as given by only the path loss.

In comparison to path loss and shadowing, fading is the interference caused by many scattered signals arriving at the receiver due to scatterers in the environment. As objects such as buildings cause a signal to scatter, this causes multipath propagation and the receiver ends up receiving multiple copies of the transmitted signal. The variations caused by this are generally experienced on the small time scale as the velocity of the signals tends to be large compared to the distances travelled.

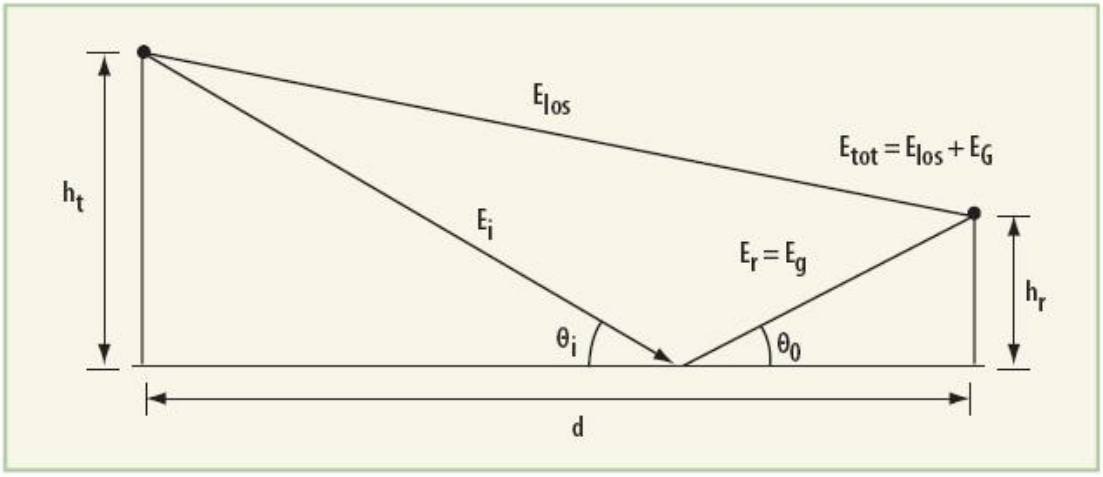

Fig 1: Two Ray Model

The relationship between Signal-to-Noise Ratio and the bit error rate experienced depends on a few factors[3]. In general the relation between p_b, the bit error rate, and the received power level P_r, follows the generic relationship shown in Equation 2.6, where N is the noise spectral density, f the raw channel bit rate, *constant* depending on the modulation and *erfc(x)* is the complementary error function shown in Equation 2.7.

$$p_b \propto erfc\left(\sqrt{\frac{\text{constant} * P_r}{N * f}}\right) \tag{2.6}$$

$$erfc(x) = 1 - \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \tag{2.7}$$

An example of this relationship for binary phase shift keying (BPSK), the bit error probability is given by Equation 2.8. In the case of this equation, the model involved in it assumed that the equation is only dependent on the distance-independent receiver noise component.

$$p_b = 0.5 * erfc\sqrt{\frac{P_r}{Nf}} \tag{2.8}$$

2.2 Models

Several types of models exist to take into account the various physical processes which affect the transmitted signal. These include the aforementioned fading and noise. Examples of noise models include the Additive white Gaussian noise and the phase noise model, while Rayleigh and Rician fading models model fading in a radio channel.

2.2.1 Additive White Gaussian Noise

AWGN channel model is one of the simpler channel models in use. In it, the only impairment taken into account is the addition of white noise, without taking into account dispersion, fading or other possible phenomena that could affect communications[19]. White noise is a random signal with a flat power spectral density.

This sort of noise is generated by many natural objects, such as the Sun and the Earth itself in the form of black body radiation. As such, the AWGN channel models the actual behavior of satellite and deep space communication links quite well, but is inadequate for terrestrial links. However, the model itself can simulate the background noise of a channel.

While white noise is considered to have the same power over all frequencies, this is not true in reality, but it is a good enough approximation for the purposes of the model. Since the bandwidth of white noise is infinite, that would mean that the power of the noise would be infinite as well, which is not possible. Since real systems have finite bandwidth though, the noise which affects them can have larger bandwidth than that of the system, which means that the noise can be considered to have an infinite bandwidth.

As the noise samples from the white noise are uncorrelated, it means that the noise affects each transmitted symbol independently. This can be seen from Equation 2.9, which is the autocorrelation function of white noise. The delta function in it means that the noise is decorrelated from its time-shifted version, which indicates the uncorrelation between samples of the white noise. Therefore, the AWGN channel is a memoryless channel.

$$R_n(\tau) = \frac{N_0}{2}\delta(\tau) \qquad (2.9)$$

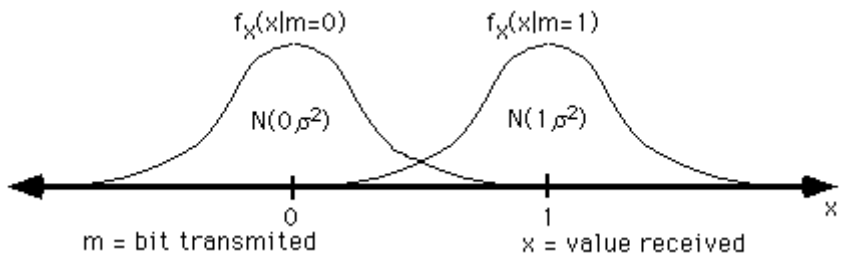

Fig 2: Representation of an AWGN-channel with noise variance of σ^2

2.2.2 Rayleigh Fading Channel

The Rayleigh fading channel model where the signal that passes through the channel is assumed to fade according to Rayleigh distribution. This model is best used when line of sight propagation is not dominant between the transmitter and receiver. It is because of this that Rayleigh fading is used to simulate propagation in an urban environment where there are many objects, such as buildings, to scatter the signal, causing multipath propagation.

The model is based on the fact that the phase of the signal received from each path can vary between 0 and 2π and the distance between the transmitter and receiver is larger than the wavelength of the carrier frequency. Based on these two facts, it is reasonable to assume that the phase is uniformly distributed between the possible values and that the phases of the paths are independent of each other.

Since there are many paths that make up the received signal, we can apply the Central Limit Theorem and model the received total value as a zero-mean Gaussian random variable. The received envelope amplitude can be described by the Rayleigh probability density function (pdf) as shown below[4].

$$p(r_0) = \frac{r_0}{\sigma^2} \exp\left[-\frac{r_0^2}{2\sigma^2} \right] \tag{2.10}$$

In the above, r_0 is the magnitude of $z(t)$, which is shown below and σ^2 is the predetection mean power of the multipath signal.

$$z(t) = \alpha(t) e^{-j\theta(t)} \tag{2.11}$$

$$\alpha(t) e^{-j\theta(t)} = x(t) + jy(t) \tag{2.12}$$

$$r_o(t) = \sqrt{x_r(t)^2 + y_r(t)^2} \tag{2.13}$$

And in this, $\alpha(t)$ is the resultant amplitude of the net received envelope and $\theta(t)$ is the resultant phase, $x(t)$ and $y(t)$ are the orthogonal components of the reflected signals and $x_r(t)$ and $y_r(t)$ being the combined received reflected components at the receiver, with the assumption that no particular received component is dominant over the others at the receiver.

2.2.3 Rician Fading Channel

A Rician fading channel is similar to a Rayleigh channel, except that there is a dominant line-of-sight (LOS) path between the transmitter and receiver, something that does not exist in a Rayleigh channel model. Besides this LOS path, there are also a number of independent paths, similar to Rayleigh. We can therefore say, that Rayleigh fading is a special case of Rician fading, which is a more generalized model. The received envelope amplitude in this case has a Rician pdf as shown below[4].

$$p(r_0) = \frac{r_0}{\sigma^2} \exp\left[-\frac{(r_0^2 + A^2)}{2\sigma^2} \right] I_0\left(\frac{r_0 A}{\sigma^2} \right) \qquad (2.14)$$

The meanings of variables in the above are identical to the ones in the Rayleigh formula (2.10). Here A is the peak magnitude of the non-faded signal component and $I_0()$ is the modified Bessel function of the first kind and zero order.

2.2.4 Okumura Model

The Okumura model is an empirical model for radio propagation, based on measurements done in Tokyo in 1960. The model is used to predict the path loss and behavior of similar links as used in the measurements which were used to create the model in question. In the case of the Okumura model, the measurements were done between 200 and 1920 MHz frequencies. While it is not truly representative of modern large cities such as New York City, it is still used as a comparison to other models[5].

The model itself is divided into three categories based on terrain categories: open area, suburban area and urban area. The first two categories are based on the third, which represents a city or a large town with large buildings consisting of two or more stories. It should therefore be easy to understand why the model may not apply exactly to a city such as the modern New York City, where many large blocking structures, such as skyscrapers, exist, unlike in 1960's Tokyo.

The median path loss for the Okumura model is expressed in Eq. 2.15.

$$L_{50}(dB) = L_{FSL} + A_{mu} - H_{tu} - H_{ru} \qquad (2.15)$$

In this equation, L_{FSL} is the free-space loss for a given distance and frequency, A_{mu} is the median attenuation relative to free-space loss in an urban area, H_{tu} is the base station height gain factor and H_{ru} is the mobile antenna height gain factor.

2.3 Discrete Channel Models

Discrete or finite-state channel is used to describe the entire communication system between points *a* and *b*, where the output symbol sequence of the system at point *b* is related to the input of the system at point *a*, with *a* usually the output of transmitters channel encoder and *b* the input of the receivers decoder. The relationship between the input and output is affected by the various elements between the two points, such as distortion and noise[18].

The term "finite state" refers to how the channel is envisioned, as containing a certain set of states or identifiable conditions with transmission from one to the next controlled by rules, which are part of the model. Finite state models are divided into two categories called memoryless models and channels with memory.

The first category is used to model channels where transmission errors are assumed to occur with no relations to one another in time. This means that the

probability of an error for a certain symbol is not affected by what happened to previous symbols. In the case where there is no inter symbolic interference or fading and the noise is additive white Gaussian in nature, this model can be applied. For a binary system, a memoryless channel means that we simply model the bit error probability of the system.

In the second category the probability of an error is affected by transitions that occurred before it and possibly those that follow. In these cases, the system does contain fading and the correlation of the signals amplitude in time due to it causes the errors to be correlated and for the signal to experience errors in bursts. Harder to model than memoryless channels, a discrete-time Markov sequence is generally used to model the errors in the channel. In the sequence, the channel contains several different states and probabilities of transition between them and each state itself contains a set of transition probabilities between the input and output symbols. This is illustrated in Figure 3.

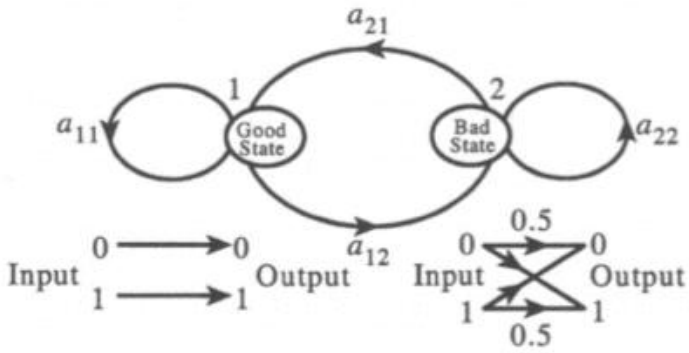

Fig 3: State transmission diagram for a two-state Markov (Gilbert) model

2.3.1 Markov Model

A Markov model is defined by a set of states, the state of the model at a specific time, set of probabilities for being in a particular state at a particular time, set of probabilities for the model transitioning from state *i* to state *j* and a set of input-to-output transition probabilities.

Consider a fading channel which will experience a deep fade from time to time. During its time out of the fade, the received signal strength is above a threshold for acceptable quality while during the fade it is below it. If we ignore the moments where the channel is in a state between those two, the situation can be described as a two-state Markov model. There is one state where the received signal strength is strong enough that the probability of an error in the transmission is very low and another where the level is so low that the probability of a transmission error approaches 0.5. As time goes on, the channel goes from one state to the other and the time spent in either state varies. The model used to represent this kind of situation is also referred to as the Gilbert model.

Another model is called the Fritchman model. In comparison to the Gilbert model which includes one "good" state and one "bad" state, the Fritchman model for binary channels divides the state space into *k* good states and *N*-*k* bad states. The good states represent error-free transmissions while in the bad ones a transmission error is always produced. Compared Gilberts state transitions which can be easily observed from Figure 3, Fritchmans state transitions are better modelled by a state transition matrix.

$$A = \begin{bmatrix} A_{GG} & A_{GB} \\ A_{BG} & A_{BB} \end{bmatrix}$$

The submatrices in the state transition matrix A represent the transition probabilities between the various possible states. This model is good to model burst errors in mobile radio channels and the parameters for the model can be estimated from empirical error distributions.

Usually, we only know or can observe the input and output of a channel and therefore the error sequence, but the state sequence is not easily observed in a state channel and it is therefore called "hidden" and the Markov model is called a hidden Markov model (HMM). Both Gilbert and Fritchman are types of hidden Markov models.

As mentioned above, the parameters for Fritchman's model can be estimated from empirical data and the same can be done for Gilbert's model as well. The program and the results talked about in Chapters 4 and 5 can be used to determine the parameters for these models. A process for estimating these parameters was developed by Leonard E. Baum and Lloyd R. Welch and it is described in Chapter 9.4.4 of Simulation of Communication Systems[18].

Chapter 3: Error Correction Methods

3.1 General

Forward Error Correction (FEC) is a type of error control technique, used in telecommunications, to detect and correct errors within the boundaries of the algorithm used to generate the error correction code. This avoids the need of retransmission inherent in Automatic Repeat reQuest (ARQ) and the subsequent "lost" time and bandwidth as the corrupted data is retransmitted. This makes Forward Error Correction a suitable technique in situations where retransmissions are either costly or impractical. It is also possible to use cryptology in tandem with error control coding (ECC) [6], but that is beyond the scope of this thesis.

FEC works by adding redundancy into the transmitted information, using an algorithm designed for it. A simple example of this is a "voting" code, where the incoming data is observed in groups of three bits. If most of the bits are 0, the transmitted data bit most likely was 0 and if most of the bits are 1 then vica versa.

FEC codes can be divided into two categories, which are convolutional codes and block codes both groups including various different codes.

3.1.1 Soft- vs. Hard-decision Decoding

Decoders can generally be divided into two classes based on how they operate, which are called soft- and hard-decision decoders.

We will concentrate on schemes that use hard-decision decoding instead of soft ones because of the inability to implement the latter type of error correction schemes in the radio system being tested. However, we will first take a quick look at

Turbo Codes, which use soft-decision decoders as part of them to achieve their impressive error correction abilities.

3.1.2 Turbo Codes

Turbo Codes are a class of very powerful error correction codes developed in 1993 by a group of French researchers and they are used in deep space communications and other low-power applications, amongst others in 3G mobile telephones. Besides low-density parity-check codes (LDPC), turbo codes are the only known practical method to almost reach the Shannon limit, as they are able to come within 0,7 dB of it [7].

Turbo Encoder

A typical turbo encoder is formed by the use of two recursive systematic convolutional (RSC) encoders that are separated by an interleaver. In the case of a turbo encoder, the interleaver has a different purpose from its usual use. Usually, the interleaver is used to scramble the code bits around various blocks in a certain pattern that can be rearranged back at the receiving end. This is done so that when the code is finally transmitted, it is not continuous in its original form and as a result from this, if the transmission encounters a burst error, the errors are spread out when the interleaving is undone and can be corrected at the receiving end.

In the Turbo encoder the interleaver rearranges the date so that the second encoder gets an permuted version of the same data that goes to the first encoder. This way, the two encoders generate different sets of code bits from the same data, providing more diversity and reliability to the transmitted code. This interleaver is generally a pseudo-random one, where the new position of a bit is decided based on its original position and on a prescribed algorithm and operates on a block basis, interleaving N bits at a time.

As the name itself implies, the encoders inside the turbo encoder are recursive, meaning that the output of the coder is used along with the input to form the

output, as can be seen in Figure 4. Convolutional codes, as mentioned in Section 3.3, encode *k*-bit blocks into *n*-bit blocks, where *k* is usually equal to 1 or 2. If the output of the encoder depends on the input bits from *m* instants of time, *m* is said to be the constraint length of the code.

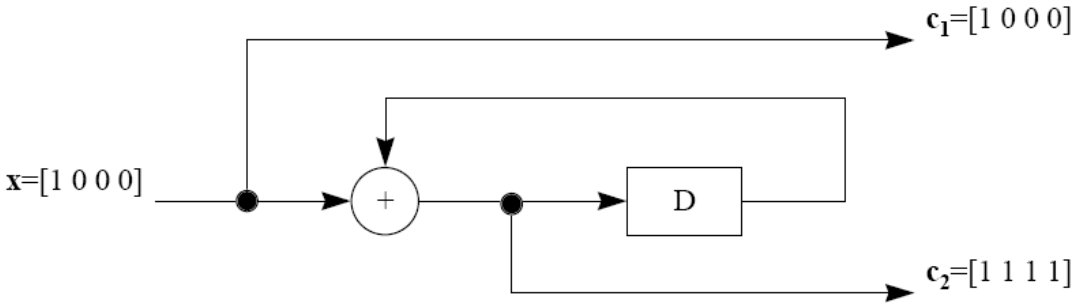

Fig. 4: An example recursive convolutional encoder

Turbo Decoder

The structure of the decoder can be thought of as a transposed version of the encoder. It is built of the same number of component decoders as there are encoders in the transmitter, connected in series and uses an interleaver as well.

The decoding process involves the decoders working together by iteratively exchanging soft-decision information. The decoders receive input from three sources: the received data block, the coded bits from their respective encoders at the transmitting end of the communications line and the estimated solution of the other decoder(s). Based on these inputs, an estimation of the message information is formed and sent ahead. This exchange of information and re-estimation continues between the decoder(s) for a number of iterations until a hard decision is made on the last soft decisions. This is different from more traditional decoding of convolutional codes where a hard or soft decision is made after the first iteration. It is this iterative soft-decision process that gives the Turbo code its near Shannon limit performance and the feedback structure from which it gained its name as an analogy to turbo engines.

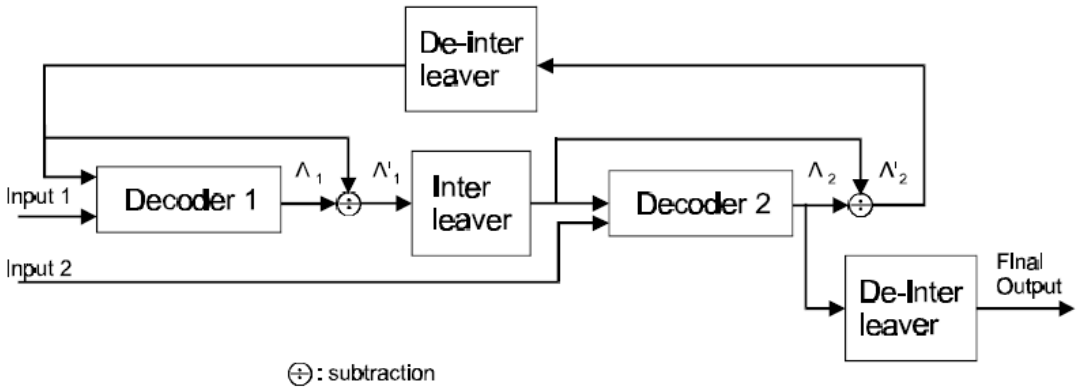

Fig. 5: Turbo decoder

While the Viterbi algorithm is an optimal choice for convolutional codes, it is unable to calculate the *a posteriori probability* (APP) for the data bits, so it cannot be used here. Instead a modified Bahl, Cocke, Jelinek and Raviv (BCJR) algorithm [8] is used, which can maximize the APP of the bit in question.

3.2 Linear Block Codes

In block codes, the information bits are divided into message blocks of *k*-bits each and after this the encoder transforms this into an *n*-bit code. When *n*>k, then *n-k* redundant bits are added to each code word, which then provide the code with the capability to combat noise and errors. These are used in the normal way of declaring, for example, what type of Hamming code is being used or discussed, in the format of calling the code a Hamming(n,k) code. For example, if we are talking of a Hamming code that codes four data bits into seven, adding three parity bits, the code is question is a Hamming(7,4) code. How to choose the suitable coding for the redundant bits is a difficult task. For example, Reed-Solomon codes are popular block codes.

3.2.1 Hamming Codes

The Hamming codes are named after their inventor, Richard Hamming and make use of a Hamming matrix. Originally published in 1950, the Hamming code is a distance-3 linear block code that was much more effective for its time than any other error-detection code in the same overhead of space and has the parameters (n,k,d). Distance in this case refers to the number of symbols that are different between two strings of equal length. For example, the distance between the binary strings 010 and 111 is two. Using suitably placed parity bits, Hammings code is able to detect two-bit errors and even correct any single-bit error. For binary Hamming codes, the length of the codeword n used is given by Equation (3.1), where the number of parity bits is r.

$$n = 2^r - 1 \tag{3.1}$$

$$k = 2^r - 1 - r \tag{3.2}$$

The parity-check matrix **H** used in the Hamming code is formed by combining an r x r identity matrix **I** with a submatrix **Q** that is made up of k columns which all are r-tuples of weight 2 or more in a manner described by Equation (3.3). The matrix **H** should contain all possible columns comprised of r bits, except a column of all zeros.

$$\mathbf{H} = \left[\mathbf{Q} \mid \mathbf{I}_r\right] \tag{3.3}$$

As the name itself implies, this matrix is used at the receiving end to perform parity checks on the incoming data and see if the data block has come through without errors. The codewords themselves can be created using another matrix, aptly called the generator matrix **G**. This matrix is formed using a transpose of the **Q** matrix and a k x k identity matrix as shown in Equation (3.4), which means that **G** is formed from a part of **H** or alternatively vice versa.

$$\mathbf{G} = \left[\mathbf{I}_k \mid \mathbf{Q}^T\right] \tag{3.4}$$

In this generator matrix, the identity matrix part represents the data partition of the coded message while the **Q** matrix represents how the parity bits are created. Below is an example of the process of creating a single code block and checking it, using a Hamming (7,4) code generation.

Assume a data sequence of four bits, 1010. The generator matrix then codes the data, but first it needs to be formed. Since the **Q** matrix represents the parity bits, decision needs to be made how to code the parity. Assume that the parity bits *p1-p3* are defined according to the data bits *d1-d4* in the following manner.

$$\begin{aligned} p1 &= d1 + d2 + d3 \\ p2 &= d1 + d3 + d4 \\ p3 &= d2 + d3 + d4 \end{aligned} \qquad (3.5, 3.6, 3.7)$$

Following this definition, the generator matrix would look like:

$$\mathbf{G} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$

The eventual codeword is gained by the product modulo 2 of the data bits, set in this example in a 1 x 4 matrix **D,** left multiplied with the generator matrix, resulting in a 7 bit codeword 1010001. This is the bit sequence that gets transmitted and at the receiving end, it is checked by the **H** matrix.

$$\mathbf{H} = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

Another multiplication is performed then (with entries modulo 2 again) with matrix **H** on the left and a vertical matrix containing the received code block on the

right. If the message was received without errors, the product of these two matrices, known also as a *syndrome*, is a null vector. This is caused by the fact that after the data has been multiplied by **G**, the resulting codeword is in the kernel of **H** and therefore, as long as it remains unmodified by errors, the multiplication will result in a null vector.

However, if a single bit error has occurred, the result is different and can be recovered from. Let's assume an error has occurred in the second bit and we have received a sequence of 1110001. The sequence of bits is multiplied with **H** once again, resulting in a syndrome of $(1\ 0\ 1)^T$. This would indicate an error within the data bits of the first and third parity bits, but not the second and looking at what the data bits these parity bits observe, we are left with an error in the second data bit and this can then be flipped and corrected. This applies the same way if one of the parity bits themselves is flipped, in which case the syndrome would have a single 1 in it or if data bit 3 is flipped, in which case the syndrome would be all 1's as it is covered by all the parity bits.

If the transmitted message experiences two bit errors, this can be detected, but as stated previously, not be recovered from anymore. Let's assume that now the first and second bits have experienced bit errors and the received message is 0110001. The resulting syndrome would be $(0\ 1\ 1)^T$, which would indicate an error in the fourth data bit. The data bit would be flipped and we would end up with three errors in our message. This same result occurs with any other combination of two bit errors. The errors are detected, but because of the decoding algorithm, any two bit error pattern will result in a seven bit sequence that contains three errors.

In other literature sources the equations for the generation of the **H** and **G** matrices might be different, for example, the placement of the identity matrix and **Q** matrix could be flipped. This is just because the placement of the data and parity bits in the encoded transmission are in different positions. Switching the placements of the rows around will merely result in a different order of bits.

The nature of the Hamming code leaves it vulnerable to burst errors as these tend to corrupt several bits in order, leaving the decoder unable to recover the

correct data, however the situation can be redeemed and the effectiveness of the code enhanced when combined with other methods, such as interleaving, as this would spread the errors around and result in more blocks containing just one error that can be recovered from. However, nowadays Hamming codes do not see any real widespread use due to their ability to only correct a single error and have been replaced by other codes.

3.2.2 Cyclic Codes

Cyclic codes are a subclass of linear codes with a few attractive qualities. Their encoding can be implemented using shift registers with feedback connections easily and there are various practical methods for decoding them. An (n,k) linear code C is called a cyclic code if every cyclic shift of a code vector in C is also a code vector in C.

A very popular and widely used class of cyclic codes are the Bose, Chaudhuri and Hocquenghem (BCH) codes, which were discovered by Hocquenghem in 1959 and by Bose and Chaudhuri independently in 1960.

A binary BCH code exists for any positive integer m and t, where $m \geqslant 3$ and $t < 2^{m-1}$ and it will contain the following parameters:

$$n = 2^m - 1 \tag{3.8}$$

$$n - k \leq mt \tag{3.9}$$

$$d_{\min} \geq 2t + 1 \tag{3.10}$$

Where n is the block length, n-k is the number of parity-check digits and d_{min} is the minimum distance. This is a BCH code capable of correcting any combination of t or fewer errors within a block of n digits. The generator polynomial of this code is specified in the terms of its roots from the Galois field GF(2^m).

A Galois or finite field is a field that contains a finite number of elements. As digital transmissions are coded in a binary form, the most widely used fields are the binary field GF(2) and its extension GF(2^m). More about Galois fields can be read from Chapter 4.13 of [9] and Chapter 2 of [10].

Reed-Solomon Codes

The R-S codes are used in several commercial applications, such as CDs, DVDs and Blue-ray, in WiMAX and DVB broadcasting systems.

The Reed-Solomon codes were introduced by Irving Reed and Gus Solomon, in 1960 [11]. They are non-binary cyclic codes and a subfamily of BCH codes.

The original description of the Reed-Solomon codes is obtained by oversampling a polynomial constructed from a set of data points. The polynomial, which is created by mapping the message to be sent to it, is evaluated at several points and these values are then the sent as the transmitted code. The idea behind this coding method comes from algebra, that any k distinct points uniquely determine a polynomial of, at most, degree k-1. Therefore, the polynomial constructed is oversampled so that even if the transmission experiences corruption, as long as sufficient amount of values are received correctly, the receiver can recover the original polynomial and therefore, decode the original data.

Like in the case of the Hamming code, the type of R-S code used can be declared using the (n,k)-format, where

$$n = 2^m - 1 \tag{3.11}$$

$$k = 2^m - 1 - 2t \tag{3.12}$$

In this case, t is the symbol-error correction capability of the code and is related to n and k according to Equation (3.13). For example, a very common and popular R-S code is (255,223), which is capable of correcting up to 16 symbol errors per block. The fact that each symbol is made up of m=8 bits, aka a byte, has probably helped this particular code to achieve popularity.

$$t = \left\lfloor \frac{n-k}{2} \right\rfloor \tag{3.13}$$

As discussed by Reed and Solomon in their original paper[5], one reason why their code became popular is its suitability in applications where errors occur in bursts and not just as flips of single bits. This is perfectly suitable for CD's, which became the first mass-produced application to use the code, as most common cause of errors were caused by scratches in the surface of the disc, which caused burst errors while reading the disc. This also applies to the use of radio communications, where transmissions can experience burst errors from various reasons.

Why is this true? Using the (255,223) code as an example, since n - k = 32, we see from Equation (3.13) that t is 16. Therefore, this code can correct up to 16 symbol errors in a block of 255. Now should the signal experience a burst of noise that was strong enough to disturb data for 121 consecutive bits, the noise would damage exactly 16 symbols within a block of data. The decoder will then correct the errors as the block arrives. It would repair any of the 16 symbol errors, despite the amount of errors it would contain. This means that to the decoder, it is irrelevant if the byte contains only a single erroneous bit or if the entire byte has been flipped. So, assuming that the start of the error burst matched the start of a symbol exactly, it would not matter if the code had experienced a 128 bit long error burst, or just 16 error bits, spread out amongst the 16 symbols. This example is not very likely to happen in real life though and is just to illustrate the workings of the code, as the chances of 121 consecutive bits inverting themselves due to noise are extremely low.

It is worth noting however in the previous example, that should this 121 bits of errors have occurred in a more random manner than in a single continuous burst, more than 16 symbols would have experienced errors by necessity, 121 at maximum and the (255,223) R-S code would have been unable to recover from this. This just goes to show that Reed-Solomon code is suitable for situations where the expected errors are burst errors by nature, but results in poor performance where the errors are random single bit errors in nature.

But how does a non-binary code work in a digital system? The previously mentioned finite fields are used to accomplish this. In the case of R-S codes, symbols from GF(2^m) are used in the construction of the codes, with the binary field GF(2) being part of this field.

3.3 Convolutional Codes

First introduced in 1955, in convolutional coding, the encoding of a *k*-bit input block doesn't only depend on the information bits themselves, but also on previous symbols[20]. Because of this, the encoder needs to have some memory registers for its use. The *k*-bits are coded into an *n*-bit output block, where *n* is greater than *k*.

The duty of the decoder is to find the most probable sequence of data based on the received sequence of bits. The first type of decoding used in convolutional codes was known as sequential decoding, originally proposed by J. M. Wozencraft, which suffered from buffer overflow and nongraceful degradation due to its large memory requirements. Nowadays, Viterbi algorithm is widely used in various applications because of it provides good maximum likelihood performance. The algorithm allows us to solve equation 3.14, where the solution is termed "maximal likelihood solution".

$$y^* = \arg\left(\underbrace{\max}_{y} P[r \mid y] \right), \qquad (3.14)$$

Where y^* is the most probable sequence of bits, r the code bits observed through noise and y is a possible sequence of bits.

3.4 ARQ

Automatic-Repeat-reQuest is an error control technique used in data communication systems to provide good system reliability. It is a simple method, easy to use and can give good results when a proper code is used for error detection, but has a few drawbacks that make it less suitable than other methods within modern communications.

ARQ communication systems require a two-way channel and an error-detecting code for them to function as designed. If this code was an (n,k) linear block code, it would add n-k parity-check bits to the k information bits to form a codeword. This codeword is then transmitted to the receiver and may experience transmission errors on the way, depending on the channel noise and other factors. The receiver then calculates the syndrome of the received codeword and if the syndrome is zero, then the received codeword is a correct codeword from the code being used. As the codeword appears to be a correct one, it is assumed to be free of errors and is accepted and a positive acknowledgement (ACK) is sent back to the sender.

If the syndrome is non-zero, the presence of errors has been detected. The receiver will transmit a negative acknowledgement (NAK) back to the sender, disregarding the received codeword and wait for a retransmission. The exact details of how the transmission and retransmission work depend on the type of ARQ scheme in use. There are three different schemes: stop-and-wait, go-back-N and selective-repeat.

3.4.1 Stop-and-wait

In the stop-and-wait scheme, the transmitter transmits a codeword and then waits for an acknowledgement from the receiver. If it receives a positive acknowledgement, the transmitter then proceeds to send the next codeword in the queue and waits once more. If a negative one is received instead, the transmitter resends the last codeword and continues to resend it until a positive acknowledgement is received.

While this scheme is very simple to implement, there are several flaws in it that make it less than ideal. The idle time spent waiting for an acknowledgement from the receiver is time wasted, making the scheme inherently inefficient. This inefficiency is all the more apparent when used in a situation where there is a large round-trip delay between the two ends of the communications line. This can be remedied by increasing the length *n* of the block used, so that the time the transmitter is transmitting is increased, while the idle time between transmissions remains the same. However, this is not a true solution as the probability that the larger block contains errors is increased with its length, causing the frequency of retransmissions to increase. There also may be limits to how much the block length can be increased, depending on the application and data format the scheme is being used in.

3.4.2 Go-back-N

In Go-back-N scheme, the transmitter continuously transmits codewords and stores them in memory to wait for either a positive or a negative acknowledgement from the receiver. As long as the acknowledgement is a positive one, the transmitter will continue to transmit continuously, but if the reply from the receiver is a negative, it stops transmitting new codewords. After the transmission of a codeword, it takes some time for the reply to arrive and this delay is called the round-trip delay and during this interval, N-1 other codewords have also been transmitted. The transmitter goes back to the codeword that the NAK indicated didn't come through error free and proceeds to retransmit it, as well as the N-1 succeeding codewords, whether they actually arrived

containing errors or not. The receiver itself discards the erroneous codeword and the *N*-1 following codewords and starts to receive the codewords again. The retransmission continues until the original erroneous codeword is received properly and an ACK is sent to the transmitter.

While this scheme improves on stop-and-go by removing the idle time in the transmitter, it's main drawback is that whenever an error has been detected in a codeword, *N*-1 other codewords are rejected as well, even though they all may by error free. This can result in great reduction in throughput performance, especially if the round-trip delay in question is long. Therefore, the go-back-N scheme starts to quickly become inefficient when high data rates and long round-trip delays are involved as more and more error-free codewords are being retransmitted. Performance is improved compared to the stop-and-wait scheme though.

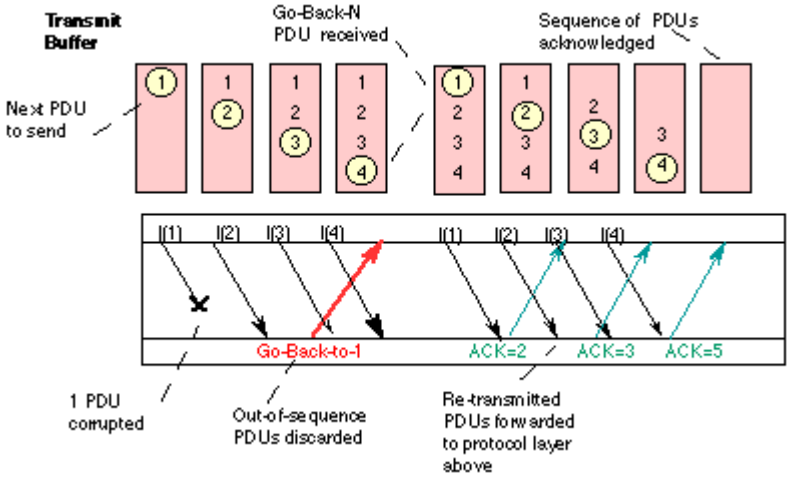

Fig 6: Go-Back-N scheme

3.4.3 Selective-repeat

The selective-repeat ARQ scheme addresses the main problem of the go-back-N scheme. In it, the transmitter also transmits continuously, but in the case of a NAK, it only resends the codeword that the negative acknowledgement refers to and then continues to transmit codewords from where it had left off. Because of this, so that the codewords can be then rearranged into their correct order, a buffer must be placed at the receiver where it can store received error-free codewords after it has received an erroneous one. When the originally erroneous codeword has been received successfully, the receiver can then release the stored codewords from the buffer in the correct order until the next error-laden codeword is encountered.

This scheme does away with the unnecessary repetition of codewords and therefore improves on throughput; it does however require large enough buffers to store codewords while waiting for the retransmission of an erroneous one. As the transmitter continues to transmit new codewords from its own buffer, if it so happens that the first retransmission arrives with errors in it as well, the need for space in the buffer continues to grow. If insufficient, there is a danger of buffer overflow and a loss of codewords

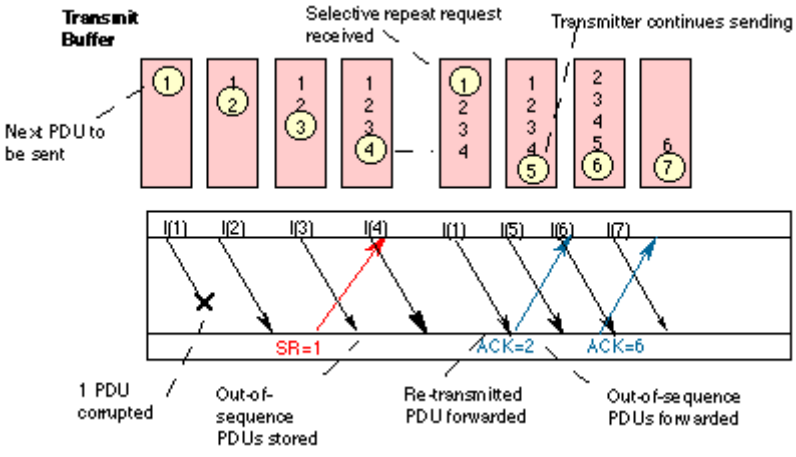

Fig 7: Selective repeat scheme

3.5 Hybrid ARQ

Hybrid ARQ is a variant of the normal ARQ, where in addition to the error-detection bits, FEC bits are also added using, for example, Turbo code. Using this combination, the HARQ performs better than ARQ when the conditions for the signal are poor, but requires more resources, resulting in lower throughput when the conditions are improved[12]. In general, HARQ schemes can be classified into two categories, called type-I and type-II schemes.

3.5.1 HARQ Type I

Type I HARQ scheme is the simpler and more straightforward version of the two. The scheme uses a code that is designed to both correct a small number of errors as well as to detect a larger number of errors. When the receiver receives a codeword and detects that it is in error, it first attempts to use the code to correct the codeword. After this, the syndrome of the codeword is calculated and if it is zero, the codeword is approved as being correct and in the case that the syndrome does not equal zero, a request for a retransmission is sent. This combination reduces the amount of retransmissions by having some error correction capability, while being able to request for retransmission in the case more than a few errors have occurred.

Because the scheme includes error correction capability as well as error detection, it requires more parity-check bits than a pure ARQ scheme, which includes only error detection to determine the need for a retransmission. Because of this, the overhead of a Type-I HARQ system is larger than that of a pure ARQ system and due to this increased overhead, the throughput of the HARQ system is lower compared to corresponding ARQ systems when the channel is good and the error rate is low. However, when the signal quality crosses over a certain point, the type-I HARQ system provides higher throughput, as the error-correction capability reduces the frequency of retransmissions.

This makes the Type-I HARQ best suited for communication systems with fairly constant levels of noise, where the coding used can be tailored to correct the majority of the received errors, reducing the need for retransmissions and improving the throughput.

3.5.2 HARQ Type II

The Type-II HARQ scheme is a more adaptive version of Type I. In the case of a nonstationary bit error rate, the Type-I has a few drawbacks. When the quality is good, error-correction is hardly needed and the extra parity bits become a waste and lessen throughput. When the channel becomes very noise, the error-correction capability may prove to be inadequate and the frequency of retransmissions increases, lessening throughput.

In the case of Type-II hybrid ARQ, the scheme behaves like an ordinary ARQ scheme, only transmitting parity-check bits for error detection added to the transmission when the channel is of good quality. This mitigates the loss of performance of Type-I HARQ when the channel is good. But when the channel becomes noisy and errors are detected in the receiver, it saves the erroneous transmission in a buffer and requests a retransmission. Unlike in other schemes though, the retransmission is not identical to the original transmission, but is formed from the original message and an error-correcting code. When this transmission of parity-check bits is received, the receiver uses it to try to correct the errors in the word that was stored in the buffer. If the correction is successful, the HARQ scheme returns to working as before. In the case that the correction is not successful, a second retransmission is requested. This may be either the repetition of the original transmission or another set of parity-check bits. The details will depend on the specific iterative scheme of the Type-II HARQ scheme and the error-correcting code used.

Chapter 4: Bit Measure Program

C++ code will be used to create pseudo random data that will be encoded and then sent through the system and received, after which the received data will be compared to the sent data to determine how many errors occurred. The code itself can be found in the appendix. The data will be created by simulating sets of 16-bit Fibonacci Linear Feedback Shift Registers (LFSR) and if need be, synchronization can be checked though the method discussed by Michal Kubíček and Michal Kováč[14]. The advantage of using LFSR is that it makes sure that there will be no short repeating sequences such as 01010101 that continue for several bytes unlike if we created bits using suitable random-functions from the C-library directly where such a possibility would exist. It also allows for easy comparison at the receiver end as it is easy to recreate the exact same sequence that was created at the transmitter end.

4.1 Measurement Program

The challenge in choosing the right type of error detection and correction scheme depends on the nature of the channel it will be used in. Several different types of models exist to simulate the nature of the various channels, ranging from Additive White Gaussian Noise (AWGN) channels to Rayleigh fading channels. While it is known what channel types generally apply to certain situations, it is never a certainty that the channel will behave exactly or close to the mathematical model used. To be able to find out how exactly a channel behaves, a program was developed to do so.

The program was coded in C++ and its aim is to create a semi-random bit sequence that can be then sent forth as it is through a channel and received. After the receiver has gained it, the transferred sequence can then be compared to a recreated version of the same sequence to find out where and how mistakes have occurred and

from this a more realistic version of the behaviour of the specific channel can be determined. A Eurocom/EIA-530 conversion board is used in-between the computer and the radio system to convert the signal to a proper format and the program in total is formed of two separate ones, each at one end of the communications line. We will next go through the operating principles of each side of the program. The code for each program has been included in the appendix.

4.1.1 Transmitter

The transmitter works by simulating the behaviour of a 16-bit Fibonacci Linear Feedback Shift Register. LFSR is a shift register whose input bit is a linear function of its previous state. As shown in Figure 8, the LFSR works by taking the values of a few of its registers and running them through a feedback formed of XOR-gates and feeding the value gained from that to the first register, while the rest of the registers values are moved forward with the rightmost registers content going to the output.

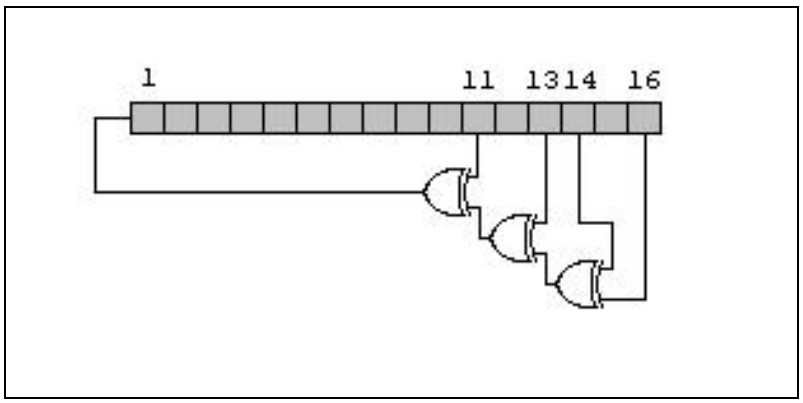

Fig. 8: 16-bit Fibonacci LFSR operation principle

The register itself exists as a static integer array called LFSR_S. The array has been defined as static so that it can be accessed outside of the main program in the code and what is done to it will be explained below. The array has been defined as having the values {1,0,1,0,1,1,0,0,1,1,1,0,0,0,0,1} in its elements, but they can be defined in the code as having any 16 bit values of your choosing, except for all zeros. This is because of the nature of the Fibonacci LFSR, which causes it to remain in a

‘locked’ state where all it ever produces are zeros if the registers in it contain just zeros at the start. Otherwise it will eventually cycle through all possible combinations of states within the 16 registers, except for the all zero state.

The next part of the program, after declaring the existence of the array is the creation of the subprogram used to manipulate the LFSR itself. The LFSR_S_fb subprogram is the feedback program of the sender for the LFSR. When it is called in the main program, it returns the content of the 16^{th} register, calculates the content of register 1 based on the contents of registers 11, 13, 14 and 16 and advances the contents of each register forward once, exactly according to how the design in Figure 8 shows.

The program was designed so that it would run indefinitely until interrupted by the user, but there was no easy way to interrupt a loop with the press of a key on the keyboard. The kbhit-command is added to achieve this. It returns a 1 or a 0 depending on if a key has been pressed or not and this can be used to interrupt a loop in the program. The exact implementation of this method was taken from an article on the Linux Journal webpage[15].

The main program of the sender program begins. Since the capability of the channel and equipment can vary, the program first asks how many bytes the program will send with single transmission iteration and how many microseconds the program will wait between these iterations. This will allow the user to determine how fast the program will send out the semi-random data generated by the 16-bit Fibonacci LFSR. It needs to be noted that because of the commands used to communicate with the Farsync-card that determining the size of a single transmission is needed, as the transmission is burst-like in nature. This can be made to resemble a continuous transmission by lowering the size of a single transmission and the period waited between transmissions to as small quantities as possible. This quantity is limited by the transmission capability of the Farsync-card itself and the transmission line. The delay is used out of practical need, as without a delay in the iterations of the loop used in the transmission, the program would simply run so fast that the registers in the Farsync-card would fill up too fast and cause overflow.

The “unsigned char transm”-array will be used to contain the final information that will be sent to the data to the Farsync-card. The format was chosen because it can easily be used to replicate the generated data from the LFSR on the bit-level. An unsigned character is formed of 8-bits, designating a particular ASCII character. Each ASCII character also corresponds to a number from 0 to 255, values generated by those bits as if they formed an 8-bit binary number. Therefore, it is easy to generate 8 bits from the LFSR and treat them as a binary number and turn those 8 generated bits into a single ASCII character and send that ASCII symbol through the channel to the receiver. This way, the actual bits sent through the channel match exactly those generated by the Fibonacci LFSR.

The binding of a socket for the use of the program is done according to the instructions found on Farsites homepage. The ”FarSync OEM RAW Sockets Application Programming Interface Reference Manual for Linux”[16] contains straightforward directions, but during the coding process, it was also noticed to contain a few errors or mistyped sections that initially caused the socket binding to fail. However, these errors were easy to notice and correct to what was presumably meant in the instructions. Of notice is that the socket is a RAW socket, which means that no protocol stacks are applied to the data sent by the kernel.

Then next is the start of the while-loop that can be interrupted by the press of any key on the keyboard, as explained above when talking about the kbhit command. The loop first generates the bits to be sent by calling on LFSR_S_fb command, filling the bits_transm array with them. These bits are then handled in groups of 8 and transformed into a single char that is added to the transm array, which was described above.

When the transm array is full, it is sent forward by the sendto command. Sendto follows the following format.

count = sendto (sd, transm, len, 0, NULL, 0)

In this format **sd** is the socket descriptor used for the connection, **msg** is a pointer to a buffer containing the data to send, **len** is the number of bytes in the buffer to send and **count** is a variable where the function returns -1 if there was an error, with the error specified in *errno*, otherwise **count** will contain the number of bytes actually sent. After the transmission, the program sleeps for the amount of time given at the start of the program and then returns to the start of the while loop.

4.1.2 Receiver

At the start of the code, the receiver contains an identical LFSR function to the one used in the transmitter end. This will be used to generate the reference data in the main partition of the program that will then be used to compare to the data we have received. A part that didn't exist in the transmitter program is command "bittivirheytys". This command can be used, if so desired, to add artificial bit errors into the received data during the time that it is being compared to the reference data. While this feature isn't necessary for any actual measurements, it can find use for testing purposes, if FEC coding or ARQ are added to the code.

Following this are identical parts to the transmitter concerning the keyboard interruption of a loop and then the main part of the program starts. The program first asks how many bytes to receive during a single iteration. This needs to be matched to what is set in the transmitter, so that the correct amount of bits will be saved without any data being missed in the reception. Next the program will ask for the desired bit error probability. If you do not wish to add any artificial errors, inputting a 0 for the probability will cause the program to skip over the error adding part during the data comparison part of the program.

Then the socket for the program is bound identically to the way it is done in the transmitter. Following this, the program will then ask for the name of the file where the received data will be saved. This is done for two purposes. First, it is saved in a file so that after the transmission is over, the program can then sort it through to find

out how errors were formed in the transmission. While it would be possible to try to do this comparison on the fly, it is also possible that with higher speeds of transmission, the checking of the data could take enough time from the computer that it would slow the receiving of the data down, resulting in an inadvertent loss of data. While some might prefer a real-time analysis of the transmission, this method avoids the possibilities of losing data in the process if the processing of the data takes longer than the period between iterations and yet gives us the same results as we would gain otherwise. Second, saving this data into a file will allow for later re-examinations of the transmission in question.

After the file has been created, the program will then start to receive data until it is manually interrupted. The data is first put into an unsigned character array named *receiv*. After the array has been filled, the data is then saved to the file after undergoing a small change in format. Two characters are handled together from the array and the 8-bit characters are changed into a single 16-bit unsigned short integer, which is identical on the bit level to the two characters and this is then saved into the previously designated file. The reason for this transformation is that a fair amount of 8-bit ASCII characters are not character per se, but some kind of control characters, such as “null”, “backspace” and “delete”. Since if we would attempt to save these kinds of characters into a file, these characters would end up either not doing anything in the file, as in the case of “null” character, or removing data already in the file, as in the case of “backspace”. In either case, there would be no evidence of these characters in the file and therefore we would receive unintended bit errors during the comparison of the data as data would have been lost during the saving of it. Because of this reason, the data is converted to numbers, which will be saved to the file without any loss. While we could have just converted each ASCII character to a plain integer matching the value of its 8-bits, an integer is composed of 32 bits. While from the standpoint of being able to retrieve the correct data we wouldn’t be affected by this at all, saving each two characters as a single 16-bit short integer does save space, making the files smaller. This should also help to speed up processes which handle these files, as there is less to handle while retaining all the necessary information.

After the receiving has ended, the file is then opened for reading and the name for a log file is asked. This log file will simply store the text information that will be displayed on the screen during the checking of the data for later use if needed.

Lastly the program begins to compare the data stored in the file to the reference data from the LFSR. The loop which checks the data file of the transmission will automatically go through it till the end of the file. The bits_receiv array will be used to store the information from the data file, while R_output will be used to house the reference data from the LFSR. During each iteration of the loop, a single output bit from the LFSR is stored into R_output, while during every iteration *i*, where *i* modulo 16 equals 0, the loop loads one number from the saved file and transforms it into 16 bits, represented in the *bits_receiv* array. At the end of each iteration, the contents of the two arrays are shifted to the left. This is because the logic behind the comparison works by considering the 32nd element of each array to be the current element under observation and the elements left of it to be previous ones.

If errors were decided to be added artificially into the data, the program then calls upon the *bittivirheytys* command. It takes the 32nd element of the *bits_receiv* array and the error probability that the user was asked to give at the start of the program. It then uses a random number generator to generate a floating point number between 0 and 1 and if it is less than the probability given for an error, the command flips that bit and returns it to the 32nd element.

Because it is possible that the connection has experienced breaks in it, causing complete loss of data for a period of time, synchronization needs to be maintained between the data from the file and reference data. If this is not done, after even a single break in the transfer of data, the reference data will out of sync and then reference data output and the received data are random to each other and 50% of the bits would be determined to be incorrect. After the program enters the 32nd iteration of the loop, it starts to check if the two sources are synchronized, in the manner described in Kubíček and Kováčs programme [14].

The program first defaults to the Resynchronize state, where it attempts to resynchronize the reference data with the data from the file containing the transmission data. It replaces the 16 elements of the Fibonacchi LFSR array with the last 16 bits read from the data file and then regenerates and replaces the last 16 output bits in the *R_output* array.

After this, the program enters the Verify state, where it compares the last 16 bits of *R_output* and *bits_receiv* to each other. If the two match completely, the data are considered to be synchronized and the program moves to the Synchronized state in the following iteration. If one or more errors are detected between the data, resynchronization hasn't been reached and the program returns to the Resynchronize status.

When the Synchronized state has been reached, the program continues to observe the last 32 bits and the number of errors detected there. It records the errors detected and their location in the log file and when synchronization is lost and then regained.

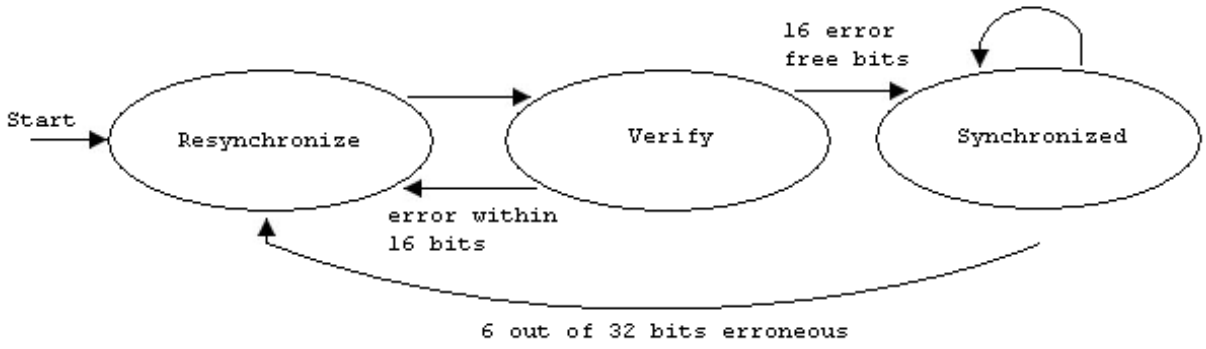

Fig. 9: Diagram of the resynchronization method and logic flow

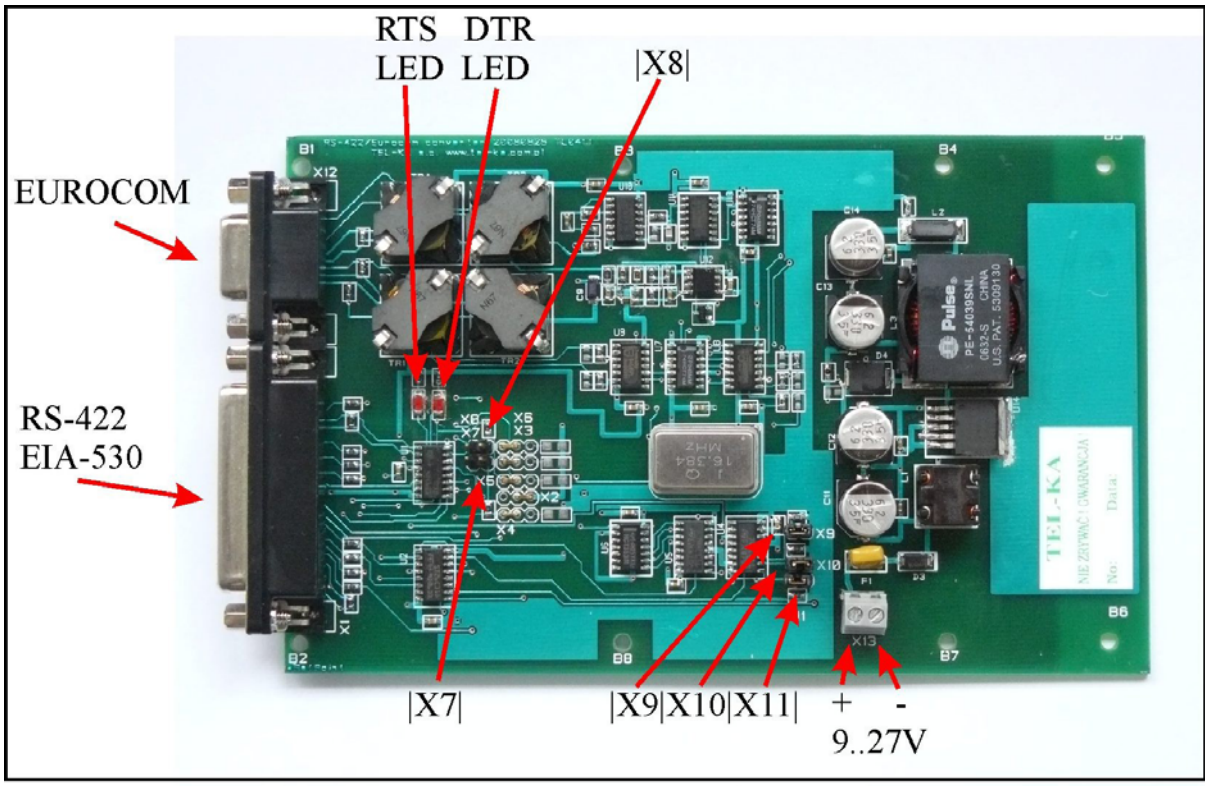

Fig 10: Eurocom/EIA-530 Converter Board Layout

Chapter 5: Measurements and Results

The measurements done were set up to test the programs functionality under actual measurement conditions and to see how the channel would behave under varying states of received signal power. No other factors were introduced into the measurements, such as possible interference from a co-channel or other sources. The computers were connected to the Eurocom converter boards, which in turn were connected to the MH 300 Radio Relays. The radios in this case were connected to each other through a cable instead of a wireless connection, with an adjustable resistor attached that could be used to control the received signal power. From the data received from the measurements, we can hopefully determine what error correction schemes would be suitable.

The equipment for the measurements consisted of:

- 2 x Radio Relay System
- 2 x Eurocom/EIA-530 Converter Board
- 2 x PC with Linux kernel
- 1 x Adjustable resistor
- Various cables

Before the actual measurements were done, the program was tested by connecting the two Eurocom/ELA-530 converted boards directly to each other and running the transmitter/receiver programs and the programs were found to be functioning as planned.

The converter board settings for jumpers X9 to X11, which can be seen in Figure 10, were set as: short, open, short. This has the effect of setting the Transmit Signal Timing and Eurocom transmit clock to use a 1024 kHz internal source. The

settings for the Farsite Farsync WAN T-Series cards inside the two PCs were made sure to be correct. The Farsync cards settings are especially important as if there are any problems with the way they are running, the programs cannot be used properly. The most important thing is that the external clock has been detected; otherwise no data will travel through. Protocol also needs to be set to raw so that no protocols are added to the data being sent. The status output of the Farsync card should look as follows if it is running properly.

```
card:          T2U FarSync WAN T-Series
ports:         2
state:         Running normally

firmware id:   5      firmware vers:  1.00.00

Configuration for port 0
physical:      X.21 (RS422/V.11)
  cable status:        Cable presence detected
  active inputs:       Indicate
  active outputs:      Control
  clock:               External, Detected
  speed:               0
protocol:      Raw packet interface
Buffer configuration:
  no of rx buffers:    8      size of rx buffers:    8192
  no of tx buffers:    8      size of tx buffers:    8192
```

The measurements were then done. The first transmission was done at the receiving power being -61 dBm to establish that the radio link was working and data could travel through it, after which the signal was dampened until connection was completely lost. Then from the data given by the program after each run, the Bit Error

Ratio was calculated for each power level. The transmissions remained error free until the received power dropped down to -96 dBm, after which the BER started to rise quite quickly as can be seen from Figure 12.

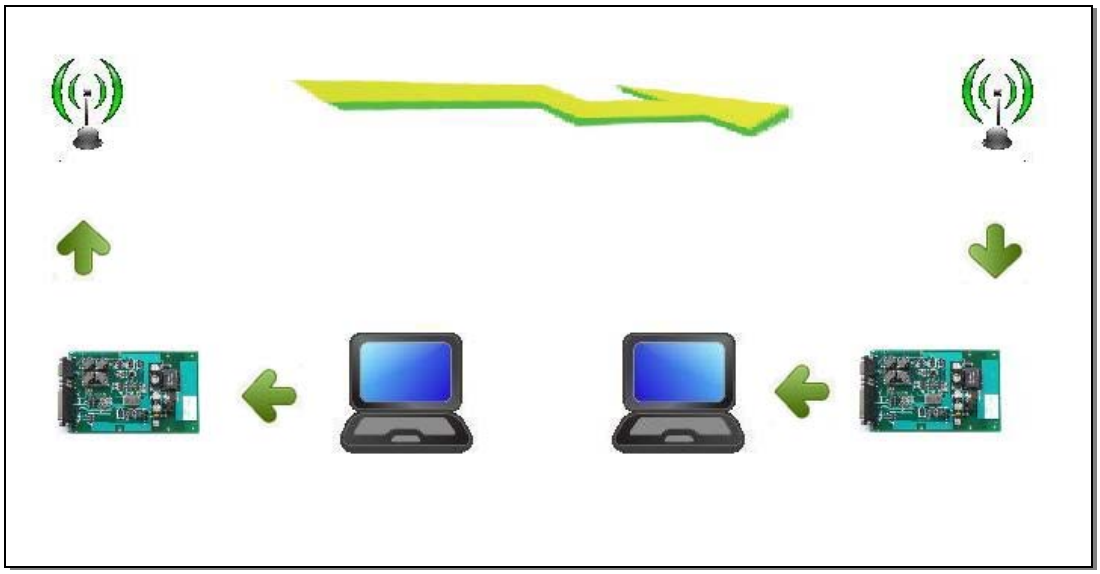

Fig 11: Channel testing configuration

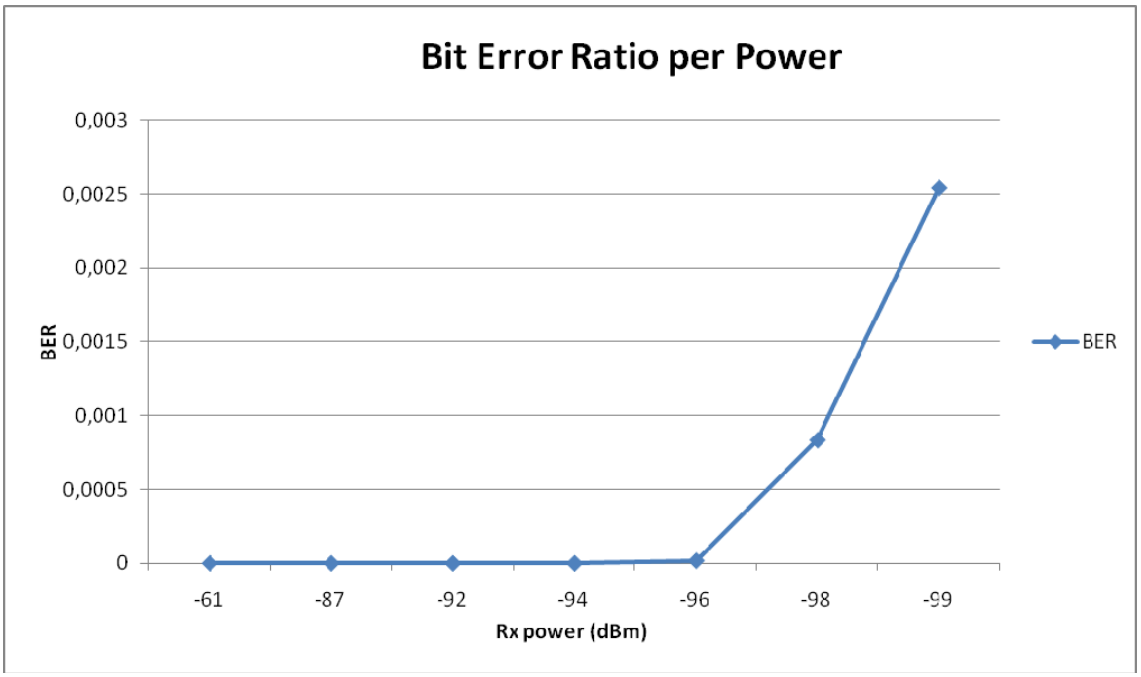

Fig 12: Measured bit error ratio by received power

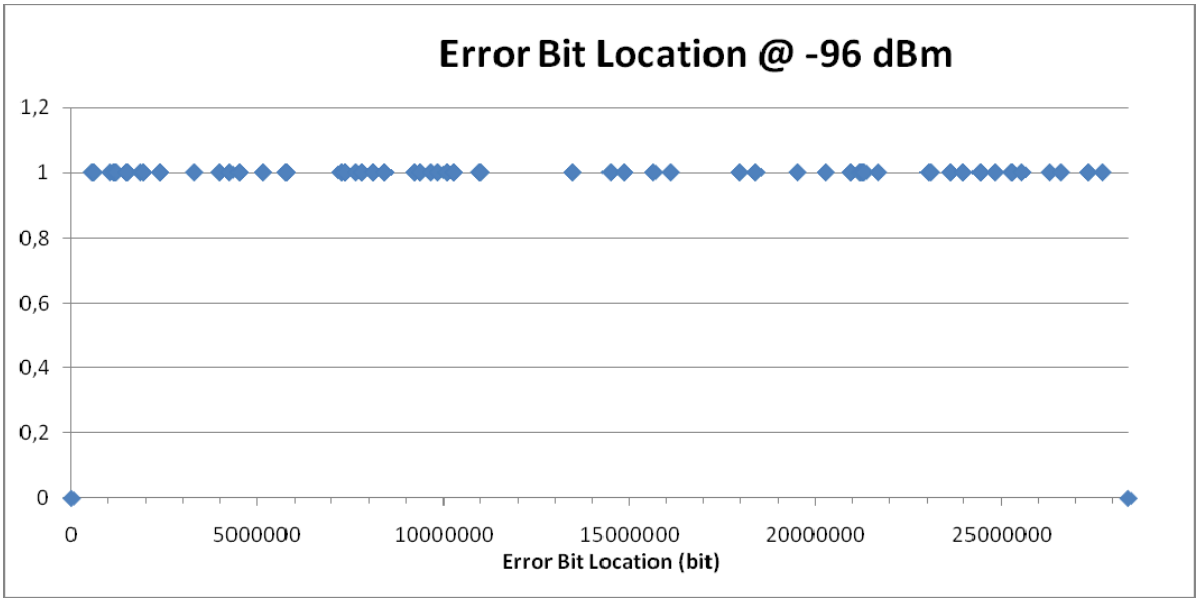

Fig 13: Error bit occurrences at -96 dBm

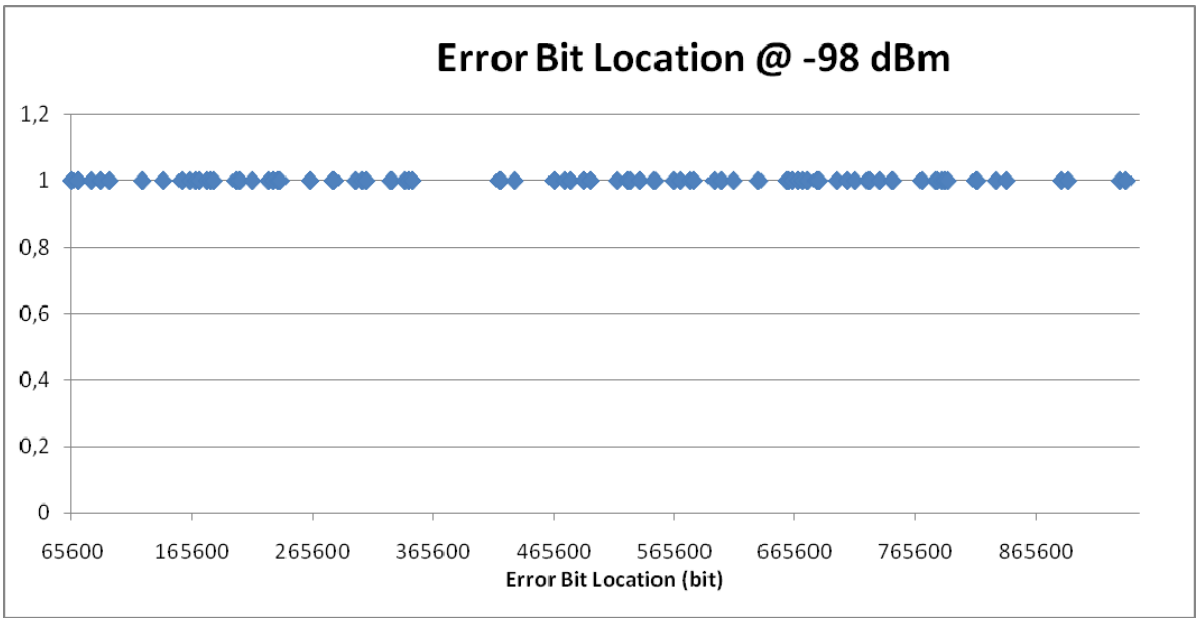

Fig 14: Error bit occurrences at -98 dBm

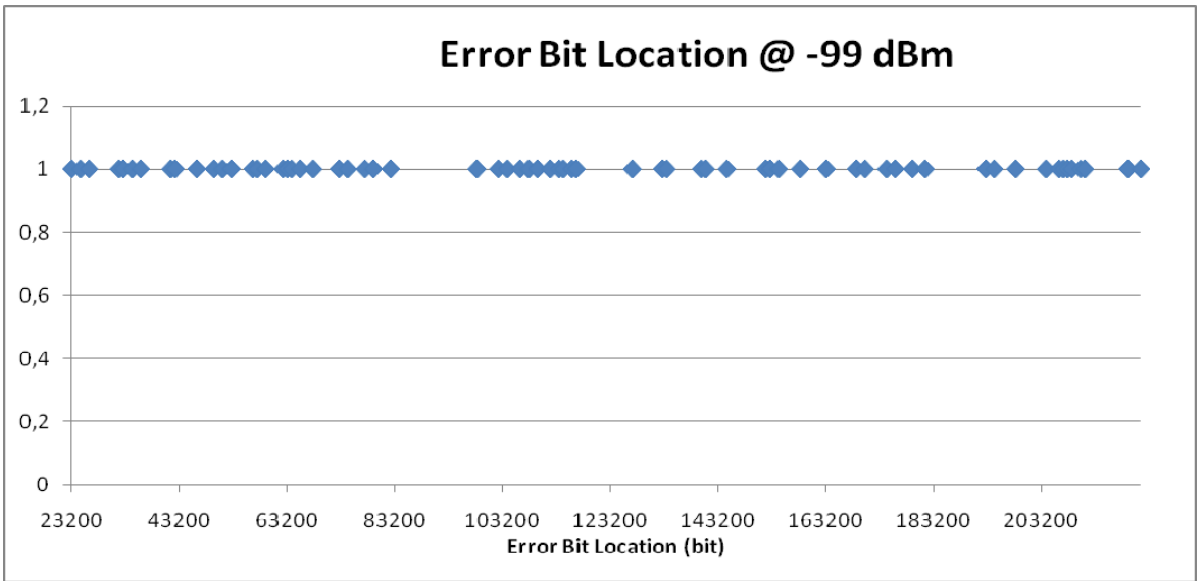

Fig 15: Error bit occurrences at -99 dBm

While the measurements at -96 dBm show the locations of the errors clearly enough, at -98 dBm the errors are already so frequent that when looking at the entire transmission, it is impossible to see any sort of patterns from it. This is true, even more so, for the -99 dBm transmission as well. Therefore, the figures of the -98 and -99 dBm transmissions show only a part of the transmission, so that any patterns can be observed from them.

From the figures above, in each transmission, we can observe a certain amount of clustering of the bit errors in all three of the transmissions with less erroneous periods in between and as the received power decreases, the frequency of the errors becomes more frequent, while remaining similar in nature. The behaviour would seem to match that of a channel with two states, good and bad, where errors occur more often during the bad state and the state switches between the two, though this may seem contradictory, as the received power level of the transmission was a constant.

Looking at the log files of the transmissions, it can be noticed that whenever the errors were detected, they were detected in groups of six, which triggers the programs resynchronization state. This happened every single time that any errors were detected and seemed to point out that there was a break in the received

transmission. To investigate this, a variation of the receivers program was created, which is quickly discussed in Chapter 6.

The program would check for errors in the same manner as the receiver, but when it detects more than 5 errors, it loads the next 32 bits from the transmission data file immediately and places them under observation. The reasoning here is to make sure that we are clearly in the part of the data after the break itself and the data being observed doesn't contain data from both before and after the break. Then the program freezes itself from continuing to load more data from the file and instead starts to move the reference data from the LFSR forward one bit at a time, comparing it to the 32 bits that were just loaded. This comparison continues until the two sets of data match, meaning that the LFSR state has caught up with the data from the file. The program keeps track of how long it takes for the LFSR to catch up, which enables us to find how long the break in the transmission was.

A clear pattern emerged from the data gained from this program. All of the breaks observed, in all of the three transmissions, were approximately 800 bits or a multiple of it in the case of the -99 dBm transmission. This would mean that as the power level of the transmission at the receiver deteriorated, it began to gather approximately 1 millisecond breaks in the data it received.

This poses a problem for determining the proper error correction coding for data to be transmitted through the channel. Error correction is ineffectual if the channel behaves so that the communications breaks before any errors actually form in the received data and only a simple scheme would be required so that the amount of missing data can be determined and a request for a retransmission sent. If the errors had not been caused due to loss of data and therefore a temporary loss of synchronization, due to the varying error levels, a Type II HARQ scheme would have been very suitable for a channel experiencing errors in the above mentioned manner.

These measurements were later replicated with slightly differing settings and measurements of datagram loss over the same channel were also performed by and

are compared here to the other results. The datagram measurements were done by using the Iperf network testing tool[17]. The results of the measurements by Iperf are shown below in Figure 16.

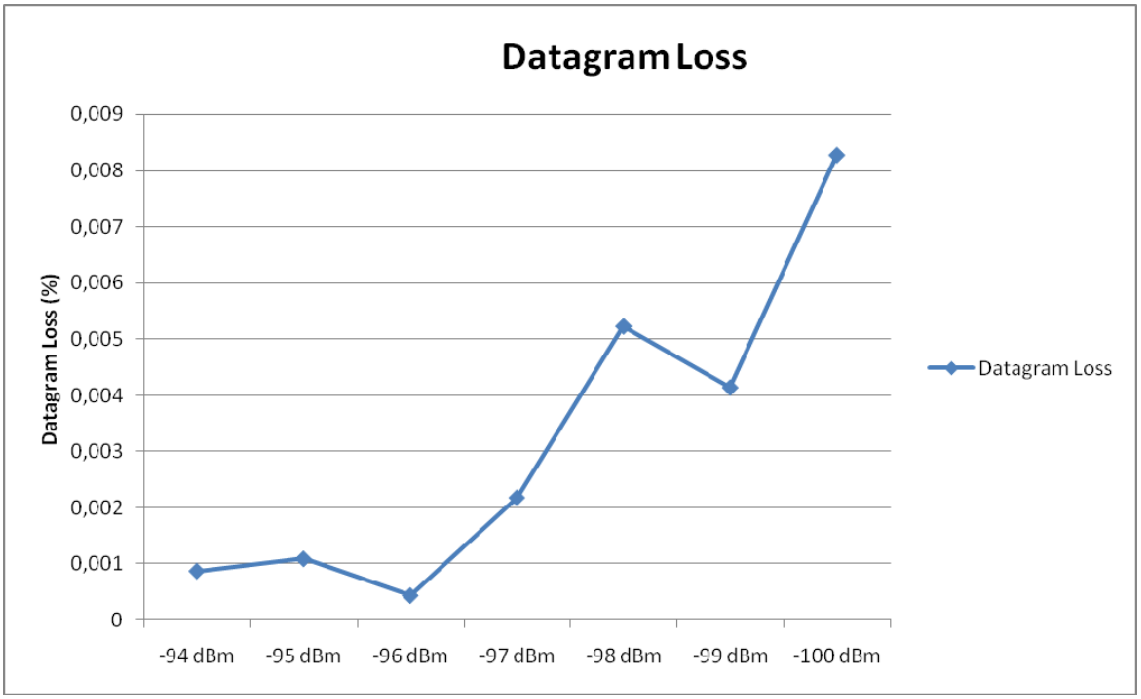

Fig 16: Datagram loss per received signal power

Seen from Figure 16, the results are not fully conclusive. At every power level measured, some datagrams were lost and the amount lost also varied in a manner which does not contribute to anything conclusive. At -96 dBm and -99 dBm, the amount of lost datagrams were actually less than in the previous measurement with greater signal power.

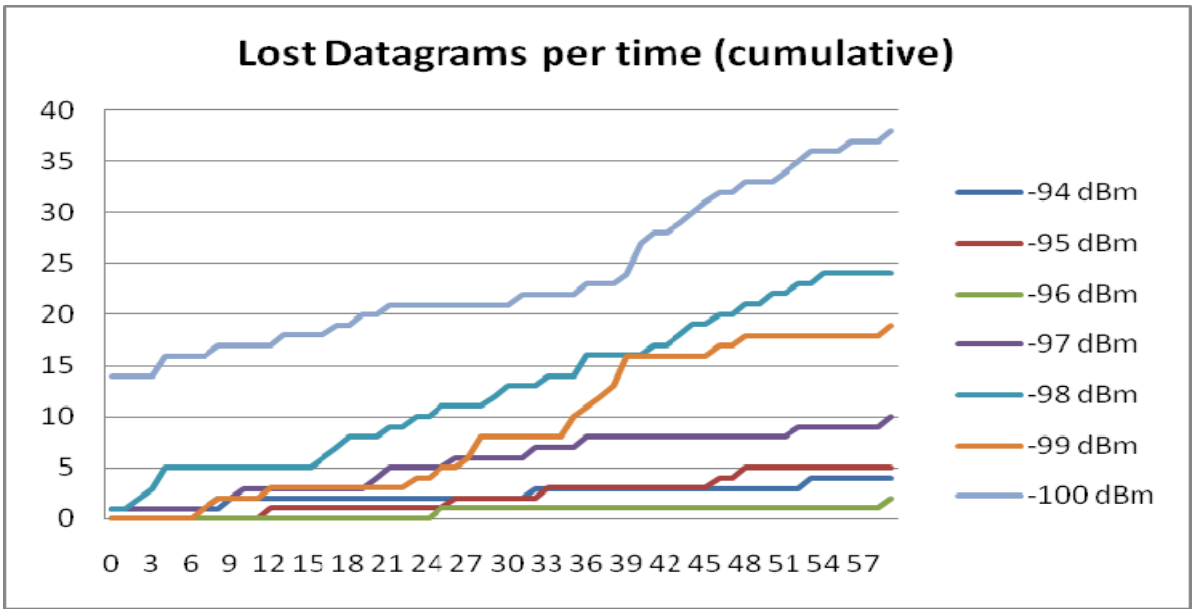

Figure 17: Cumulative Datagram Loss as a measure of time in seconds

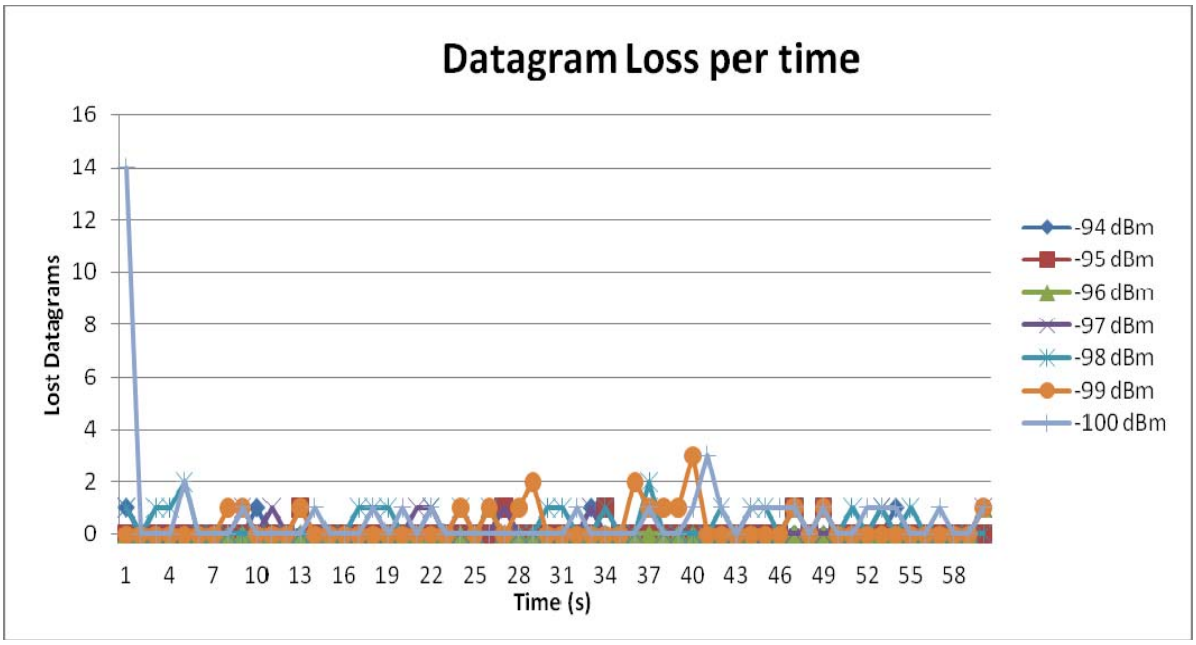

Figure 18: Datagram Loss as a measure of time

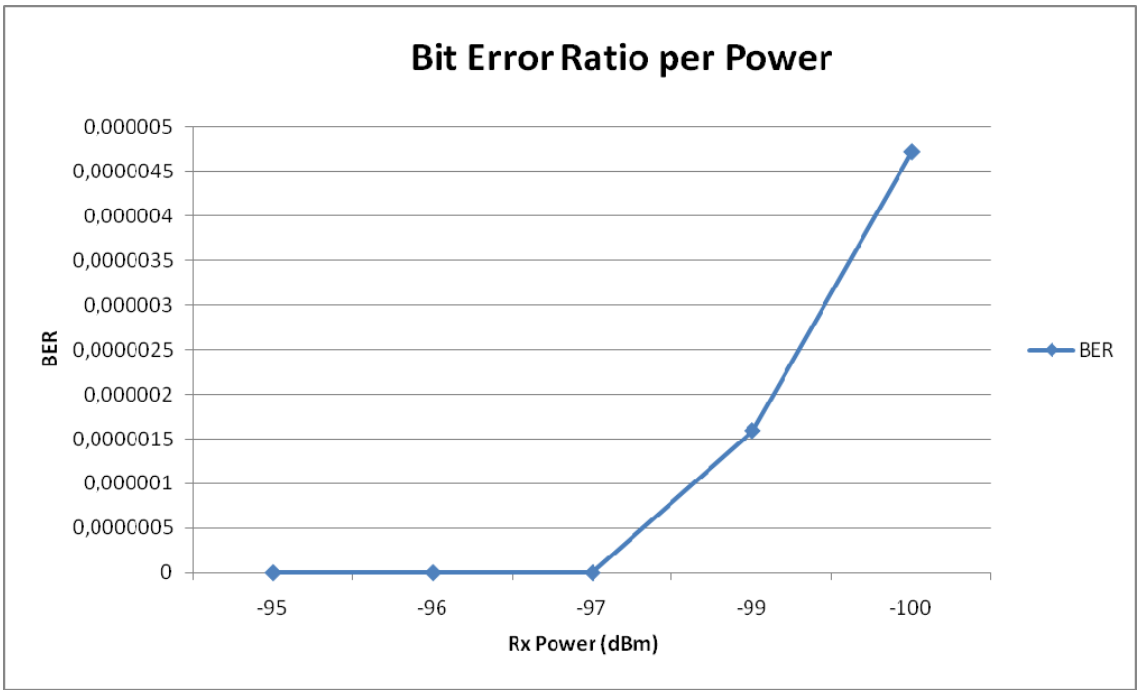

Figure 19: Bit Error Ratio per Received Power

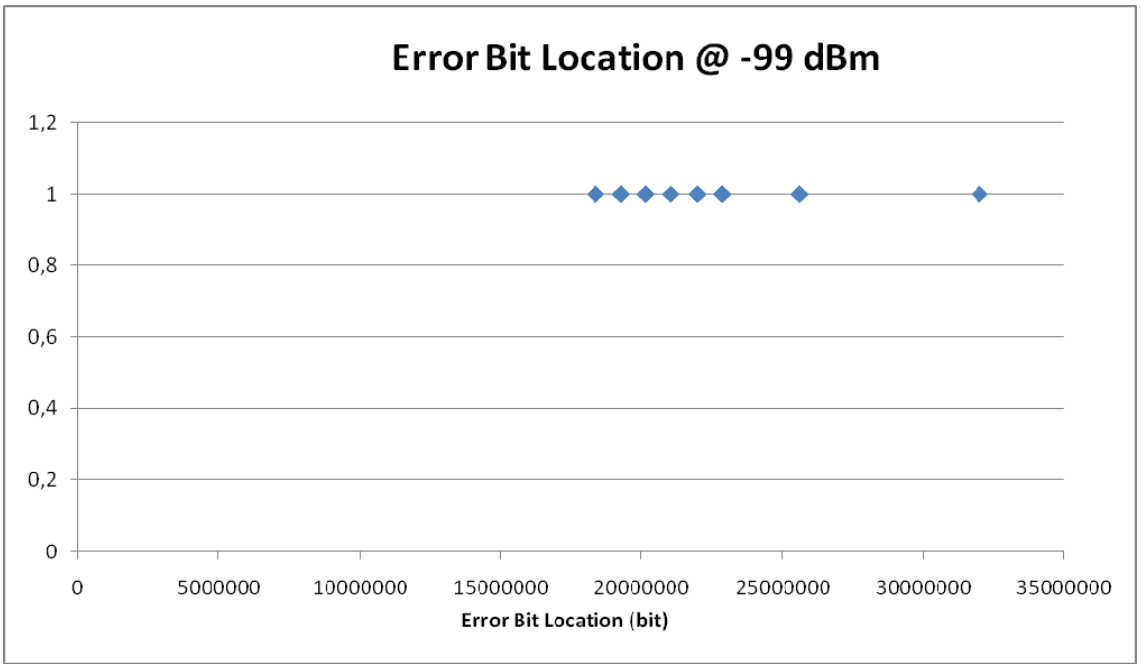

Figure 20: Error bit occurrences at -99 dBm

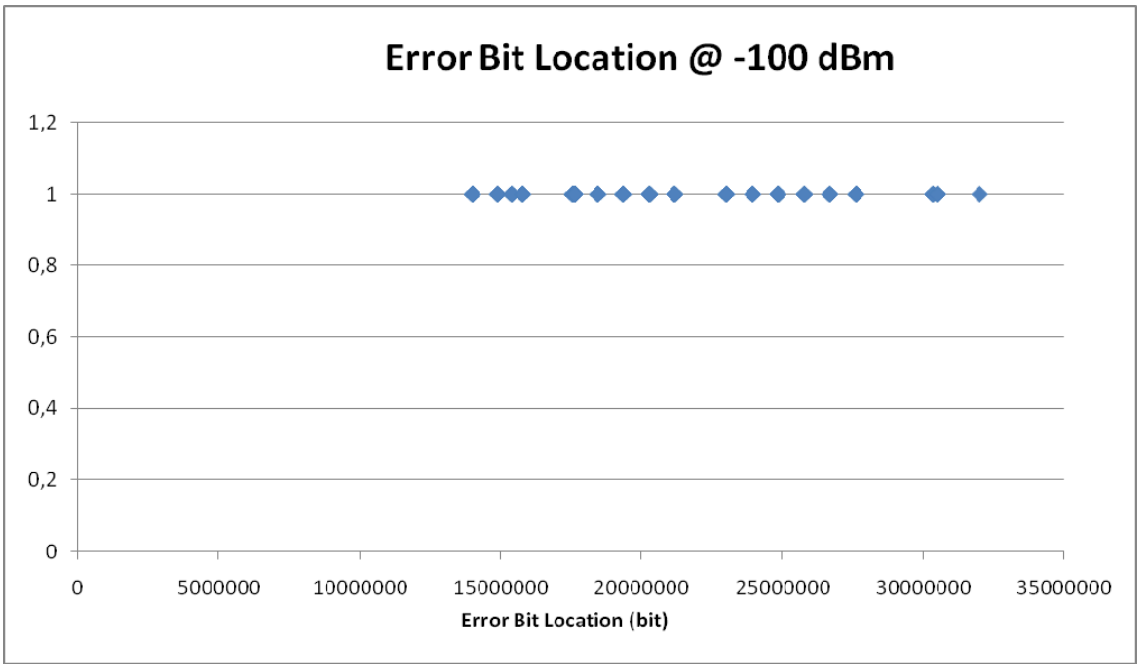

Figure 21: Error bit occurrences at -100 dBm

Chapter 6: Conclusions

As seen from the brief look in Chapter 2, there are many things that need to be considered when planning communications through a wireless channel. The environment through which the communications happens can have a variety of effects on the signal travelling through it, resulting in errors in the received data. Many error correction methods have been developed during the decades of wireless communications with various strengths and weaknesses, advantages and trade-offs, but without knowledge of the behaviour of the channel in the environment it is being used, sub-optimal choices might be made.

By using a simple program that can generate data which can be replicated exactly at the receiver, we were able to make measurements with real communications equipment indoors to determine how errors occurred in the transmitted data depending on the received power level, effectively mirroring a situation where the receiver moved further away from the transmitter, resulting in greater path loss.

The measurements were able to show us some of the behaviour of the channel as the channel seemed to experience periods of greater and lower error ratios and the overall BER increased with lower received power as was expected. The behaviour of the errors points towards a channel with memory model, mentioned in section 2.3. Roughly, the errors could be seen to occur according to a two-state Markov model. The program itself was shown to work and has the advantage that it is usable with any computer, as long as that computer can compile the program as it is coded in C++, that can be connected to a radio system. This offers a rather flexible way in which different channels can be measured and another way to determine suitable error correction methods to be used. A second program was created, which can be used to analyze the resulting data files from this program and can calculate information such as the lengths of error-free periods or erroneous periods and the amount of both without the user having to do any calculations himself.

Together these two programs offer way for the user to analyze the channel being measured with relative ease, whether you needed to measure the BER of the channel or the manner and frequency in which errors occur in it.

Bibliography

[1] “Directional Radio Channel Measurements at Mobile Station in Different Radio Environments at 2.15 GHz"; Kimmo Kalliola, Heikki Laitinen, Lasse Vuokko, Pertti Vainikainen; Proceedings of 4th European Personal Mobile Communications Conference (EPMCC2001), Vienna, Austria; ISBN 3-85133-023-4, 2001

[2] “Wireless Channel Models”, Chapter 2; Ana Aguiar, James Gross; TKN Technical Report Series, Report TKN-03007, 2003

[3]”Mobile, Wireless, and Sensor Networks: Technology, Applications, and Future Directions”; Rajeev Shorey, A. Ananda, Mun Choon Chan and Wei Tsang Ooi; Wiley-IEEE Press, ISBN 0471718165, 2006, p. 119-120.

[4]”Digital Communications: Fundamentals and Applications (2nd Edition)”, Bernard Sklar, Prentice Hall PTR, ISBN 0130847887, 2001, p. 955-956.

[5]”Introduction to RF propagation”, John S. Seybold, Wiley-Interscience, ISBN 0471655961, 2005, p. 146-152.

[6] “Physical-Layer Security: Combining Error Control Coding and Cryptology”; Willie K. Harrison, Steven W. McLaughlin; 2009

[7] “Near Shannon Limit Error-Correcting Coding and Decoding: Turbo-Codes (1)”; Claude Berrou, Alain Glavieux and Punya Thitimajshima; 1993

[8] “Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate”; L. R. Bahl, J. Cocke, F. Jelinek and J. Raviv; IEEE Transactions on Information Theory, vol. IT-20(2), 1974, pp.284-287.

[9] “Basic Algebra I “(2nd ed.), Nathan Jacobson, W. H. Freeman and Co., ISBN 978-0-7167-1480-4, 1985

[10] “Error Control Coding: Fundamentals and Applications”; Shu Lin, Daniel J. Costello; Prentice Hall, ISBN 013283796X, 1983.

[11] “Polynomial Codes Over Certain Finite Fields”; I. S. Reed, G. Solomon; SIAM Journal of Applied Math., vol. 8, 1960, pp. 300-304.

[12] “Automatic-Repeat-Request Error-Control Schemes”; Shu Lin, Daniel J. Costello Jr., Michael J. Miller; Vol 22, No. 12, IEEE Communications Magazine, 1984.

[13] “Essentials of Error-Control Coding”; Jorge Castiñeira Moreira, Patrick Guy Farrell; ISBN 047002920X, John Wiley & Sons, 2006

[14] “PRBS Test Sequence Synchronization in Bit Error Measurement of FSO Links”; Michal Kubíček , Michal Kováč; Doctoral Degree Programme (2), FEEC BUT

[15]http://www.linuxjournal.com/files/linuxjournal.com/linuxjournal/articles/011/1138/1138l1.html

[16]http://www.farsite.co.uk/manuals/FarSync_Linux_Raw_Sockets_API_Reference_Manual_V1.6.pdf

[17] Iperf site at SourceForge, http://sourceforge.net/projects/iperf/

[18]"Simulation of Communication Systems, Second Edition: Modeling, Methodology and Techniques"; Michel Jeruchim, Philip Balaban, Sam Shanmugan; ISBN 0306462672, Springer, 2000

[19]"Fundamentals of Wireless Communications"; David Tse, Pramod Viswanath; ISBN 0521845270, Cambridge University Press, 2005

[20]" Wireless Communications", Andrea Goldsmith, ISBN 0521837162, Cambridge University Press, 2005

Appendix 1: lfsr_send.c-code

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <linux/if_ether.h>
#include <netinet/in.h>
#include <sys/ioctl.h>

#include <arpa/inet.h>
#include <netdb.h>


#include<linux/if_packet.h>
#include<net/if.h>
#include<sys/ipc.h>
#include<sys/msg.h>
#include<net/if_arp.h>
#include <sys/select.h>


static int LFSR_S[16]={1,0,1,0,1,1,0,0,1,1,1,0,0,0,0,1};



int LFSR_S_fb(){
        int feedback;
        int temp1,temp2, temp3;

        temp3=LFSR_S[15];

        if (LFSR_S[15]!=LFSR_S[13]) temp1=1;
        else temp1=0;

        if (LFSR_S[12]!=temp1) temp2=1;
        else temp2=0;

        if (LFSR_S[10]!=temp2) feedback=1;
        else feedback=0;
```

```
		for (int i = 15; i > 0; --i){ LFSR_S[i]=LFSR_S[i-1]; }
		LFSR_S[0]=feedback;

		return temp3;
}



int kbhit(void)
{
 struct timeval tv;
 fd_set read_fd;

 /* Do not wait at all, not even a microsecond */
 tv.tv_sec=0;
 tv.tv_usec=0;

 /* Must be done first to initialize read_fd */
 FD_ZERO(&read_fd);

 /* Makes select() ask if input is ready:
  * 0 is the file descriptor for stdin    */
 FD_SET(0,&read_fd);

 /* The first parameter is the number of the
  * largest file descriptor to check + 1. */
 if(select(1, &read_fd,NULL, /*No writes*/NULL, /*No exceptions*/&tv) == -1)
  return 0;  /* An error occured */

 /*  read_fd now holds a bit map of files that are
  * readable. We test the entry for the standard
  * input (file 0). */

if(FD_ISSET(0,&read_fd))
   /* Character pending on stdin */
   return 1;

 /* no characters were pending */
 return 0;
}


int main(void)
{
		int m, usec;
		char dec;
		int dec_int;
```

```
		printf("\nHow many bytes to transmit during a single iteration? (Max.
1500) MUST BE AN EVEN NUMBER!\n");
		scanf("%i", &m);
		printf("\n");


		if (m<1)
				m=1;

		if (m>1500)
				m=1500;


		printf("How many microseconds will be waited between iterations? (Min.
1)\n");
		scanf("%i",&usec);
		printf("\n");




		int n=m*8;
		int bits_transm[n];
		unsigned char transm[m];

		unsigned int t=0;



int sd;
int af, type, protocol;

int len, retval, flags, count;
len = n/8;
flags = 0;
af = PF_PACKET;
type = SOCK_RAW;
protocol = htons(ETH_P_CUST);
sd = socket(af, type, protocol);

int addr_len;
struct ifreq req;
struct sockaddr_ll sll;
int ifindex;
char name[] = "sync0\0";
strcpy ( req.ifr_name, name );
```

```
/* Bind socket to a single interface. Start by getting the index */
if ( ioctl ( sd, SIOCGIFINDEX, &req ) < 0 )
{
 printf("Error getting interface %s index.\n%s\n", name,
strerror ( errno ) );
}
ifindex = req.ifr_ifindex;



/* Now bind it */
bzero ( &sll, sizeof ( sll ));
sll.sll_family = AF_PACKET;
sll.sll_hatype = ARPHRD_RAWHDLC;
sll.sll_ifindex = ifindex;
addr_len = sizeof(sll);
if ( (retval = bind ( sd, (struct sockaddr *)&sll, addr_len)) < 0 )

{
printf("Problem binding to interface %s.\n%s\n", name,
strerror ( errno ));
}




while(!kbhit()){


// create transmit signal

                for (int i = 0; i < n; ++i){ bits_transm[i]=LFSR_S_fb();

                }

// change into char-form

                for (int i = 0; i < n/8; ++i){

                                int bit=0;

                                if (bits_transm[0+(i*8)]==1) bit=bit+128;
                                if (bits_transm[1+(i*8)]==1) bit=bit+64;
                                if (bits_transm[2+(i*8)]==1) bit=bit+32;
                                if (bits_transm[3+(i*8)]==1) bit=bit+16;
```

```
                        if (bits_transm[4+(i*8)]==1) bit=bit+8;
                        if (bits_transm[5+(i*8)]==1) bit=bit+4;
                        if (bits_transm[6+(i*8)]==1) bit=bit+2;
                        if (bits_transm[7+(i*8)]==1) bit=bit+1;

                        transm[i]=(char)bit;

                }

// transmit signal



count = sendto (sd, transm, len, 0, NULL, 0 );
t++;

printf("Count: %i\n",count);
printf("Errno: %i\n",errno);
printf("Iteration: %i\n",t);

usleep(usec);
                }
}
```

Appendix 2: lfsr_recv_saving.c-code

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include <unistd.h>
#include <errno.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <linux/if_ether.h>
#include <netinet/in.h>
#include <sys/ioctl.h>

#include <arpa/inet.h>
#include <netdb.h>

#include <iostream>
#include <fstream>
#include <string.h>


#include<linux/if_packet.h>
#include<net/if.h>
#include<sys/ipc.h>
#include<sys/msg.h>
#include<net/if_arp.h>
#include <sys/select.h>


static int LFSR_R[16]={1,0,1,0,1,1,0,0,1,1,1,0,0,0,0,1};

using namespace std;


int LFSR_R_fb(){
                int feedback;
                int temp1,temp2,temp3;

                temp3=LFSR_R[15];

                if (LFSR_R[15]!=LFSR_R[13]) temp1=1;
                else temp1=0;

                if (LFSR_R[12]!=temp1) temp2=1;
```

```
		else temp2=0;

		if (LFSR_R[10]!=temp2) feedback=1;
		else feedback=0;

		for (int i = 15; i > 0; --i){ LFSR_R[i]=LFSR_R[i-1];}
		LFSR_R[0]=feedback;

		return temp3;
}


//----------------------

int bittivirheytys(int bit, float berperc) {

		float random;


		if((random=(rand()/(float(RAND_MAX)+1)))<berperc)
			{
				if (bit==1) bit=0;
				else bit=1;
			}

		return(bit);
}


//-----------------------------



int kbhit(void)
{
  struct timeval tv;
  fd_set read_fd;

  /* Do not wait at all, not even a microsecond */
  tv.tv_sec=0;
  tv.tv_usec=0;

  /* Must be done first to initialize read_fd */
  FD_ZERO(&read_fd);

  /* Makes select() ask if input is ready:
   * 0 is the file descriptor for stdin    */
  FD_SET(0,&read_fd);
```

```
  /* The first parameter is the number of the
   * largest file descriptor to check + 1. */
  if(select(1, &read_fd,NULL, /*No writes*/NULL, /*No exceptions*/&tv) == -1)
    return 0;  /* An error occured */

  /*  read_fd now holds a bit map of files that are
   * readable. We test the entry for the standard
   * input (file 0). */

if(FD_ISSET(0,&read_fd))
    /* Character pending on stdin */
    return 1;

  /* no characters were pending */
  return 0;
}



int main(void)
{

                int m;
                printf("\nHow many bytes will be received from sender during an
iteration? (Max. 1500) MUST BE AN EVEN NUMBER!\n");
                scanf("%i", &m);
                printf("\n");


                if (m<1)
                                m=1;

                if (m>1500)
                                m=1500;


                int n=m*8;
                int bits_transm[n];
                unsigned char transm[m];
                unsigned char receiv[m];

                float berperc;

                printf("Probability of bit error? (0.0 - 1.0)");
                printf("\n");
                scanf("%f", &berperc);
                printf("\n");
```

```
		if (berperc<0.0)
			berperc=0.0;

		if (berperc>1.0)
			berperc=1.0;


		srand((unsigned)time(0));


int sd;
int af, type, protocol;

int i, count;
int buflen;
buflen = n/8;


int retval, flags;
flags = 0;

af = PF_PACKET;
type = SOCK_RAW;
protocol = htons(ETH_P_CUST);
sd = socket(af, type, protocol);

int addr_len;
struct ifreq req;
struct sockaddr_ll sll;
int ifindex;
char name[] = "sync0\0";
strcpy ( req.ifr_name, name );


/* Bind socket to a single interface. Start by getting the index */
if ( ioctl ( sd, SIOCGIFINDEX, &req ) < 0 )
{
 printf("Error getting interface %s index.\n%s\n", name,
strerror ( errno ) );
}
ifindex = req.ifr_ifindex;



/* Now bind it */
bzero ( &sll, sizeof ( sll ));
sll.sll_family = AF_PACKET;
```

```
sll.sll_hatype = ARPHRD_RAWHDLC;
sll.sll_ifindex = ifindex;
addr_len = sizeof(sll);
if ( (retval = bind ( sd, (struct sockaddr *)&sll, addr_len)) < 0 )

{
printf("Problem binding to interface %s.\n%s\n", name,
strerror ( errno ));
}



time_t rawtime;
time (&rawtime);
string date;
date=ctime(&rawtime);

ofstream myfile;

string faili;
cout << "Enter name for data file: ";
cin >> faili;
myfile.open(faili.c_str());


unsigned char temp;
unsigned short int tempnum, tempnum2, tempnum3, tempnum4;



printf("Starting to read \n");




while(!kbhit()){

count = recvfrom ( sd, receiv, buflen, 0, NULL, 0 );
printf ("Read completed with a size of %d\n", count);


printf("\n");
printf("Count: %i\n",count);
printf("Errno: %i\n",errno);

for (i = 0; i< count; i++)
{
```

```
temp=receiv[i];

if (i%2==0){
tempnum3=(int)temp;
tempnum4=0;


                if (tempnum3>127){
                tempnum4=tempnum4+32768;
                tempnum3=tempnum3-128;
                }
                if (tempnum3>63){
                tempnum4=tempnum4+16384;
                tempnum3=tempnum3-64;
                }

                if (tempnum3>31){
                tempnum4=tempnum4+8192;
                tempnum3=tempnum3-32;
                }

                if (tempnum3>15){
                tempnum4=tempnum4+4096;
                tempnum3=tempnum3-16;
                }

                if (tempnum3>7){
                tempnum4=tempnum4+2048;
                tempnum3=tempnum3-8;
                }

                if (tempnum3>3){
                tempnum4=tempnum4+1024;
                tempnum3=tempnum3-4;
                }

                if (tempnum3>1){
                tempnum4=tempnum4+512;
                tempnum3=tempnum3-2;
                }

                if (tempnum3>0){
                tempnum4=tempnum4+256;
                tempnum3=tempnum3-1;
                }
```

```
}

if(i%2==1){ tempnum=(short int)temp;
tempnum=tempnum+tempnum4;

myfile << tempnum << "\n";
}


}//end of for (i = 0; i< count; i++) loop


}//end of while(!kbit()) loop


myfile.close();

ifstream myfile2;
myfile2.open(faili.c_str());
string logi;
cout << "Enter name for log file: ";
cin >> logi;
ofstream log;
log.open(logi.c_str());




// receiver side synchronization

                int SYNC_STAT=0; //0=RE_SYNC 1=VERIFY 2=SYNC
                int R_output[31];
                int BER=0;
                int counter=0;
                int bits_receiv[46];
                i=0;

while(!myfile2.eof()){



                R_output[31]=LFSR_R_fb();


                if(i%16==0){
```

```
myfile2 >> tempnum2;


if (tempnum2>32767){
tempnum2=tempnum2-32768;
bits_receiv[31]=1;
}
else bits_receiv[31]=0;
if (tempnum2>16383){
tempnum2=tempnum2-16384;
bits_receiv[32]=1;
}
else bits_receiv[32]=0;
if (tempnum2>8191){
tempnum2=tempnum2-8192;
bits_receiv[33]=1;
}
else bits_receiv[33]=0;
if (tempnum2>4095){
tempnum2=tempnum2-4096;
bits_receiv[34]=1;
}
else bits_receiv[34]=0;
if (tempnum2>2047){
tempnum2=tempnum2-2048;
bits_receiv[35]=1;
}
else bits_receiv[35]=0;
if (tempnum2>1023){
tempnum2=tempnum2-1024;
bits_receiv[36]=1;
}
else bits_receiv[36]=0;
if (tempnum2>511){
tempnum2=tempnum2-512;
bits_receiv[37]=1;
}
else bits_receiv[37]=0;
if (tempnum2>255){
tempnum2=tempnum2-256;
bits_receiv[38]=1;
}
else bits_receiv[38]=0;
```

```
                if (tempnum2>127){
                tempnum2=tempnum2-128;
                bits_receiv[39]=1;
                }
                else bits_receiv[39]=0;
                if (tempnum2>63){
                tempnum2=tempnum2-64;
                bits_receiv[40]=1;
                }
                else bits_receiv[40]=0;
                if (tempnum2>31){
                tempnum2=tempnum2-32;
                bits_receiv[41]=1;
                }
                else bits_receiv[41]=0;
                if (tempnum2>15){
                tempnum2=tempnum2-16;
                bits_receiv[42]=1;
                }
                else bits_receiv[42]=0;
                if (tempnum2>7){
                tempnum2=tempnum2-8;
                bits_receiv[43]=1;
                }
                else bits_receiv[43]=0;
                if (tempnum2>3){
                tempnum2=tempnum2-4;
                bits_receiv[44]=1;
                }
                else bits_receiv[44]=0;
                if (tempnum2>1){
                tempnum2=tempnum2-2;
                bits_receiv[45]=1;
                }
                else bits_receiv[45]=0;
                if (tempnum2>0){
                tempnum2=tempnum2-1;
                bits_receiv[46]=1;
                }
                else bits_receiv[46]=0;



                }

//  add errors
```

```
            if(berperc!=0) bits_receiv[31]=bittivirheytys(bits_receiv[31],berperc);

// add errors


            if(i>30){

            if (SYNC_STAT==2){
            int ERR=0;

            for (int k=31; k>=0; --k){
            if (R_output[31-k]!=bits_receiv[31-k]) ERR=ERR+1;
            }
            if (ERR>5) SYNC_STAT=0;

            else {

            counter=counter+1;
            int temp=0;
            if (R_output[31]!=bits_receiv[31]) temp=1;
            if (R_output[31]!=bits_receiv[31]) {printf("error bit at %i\n",i);
            log << "error bit at " << i << "\n";}
            BER=BER+temp;
            }

            }



            if (SYNC_STAT==1){
            int ERR=0;

            for (int k=15; k>=0; --k){
            if (R_output[31-k]!=bits_receiv[31-k]) ERR=ERR+1;
            }
            if (ERR>0) {SYNC_STAT=0;
            printf("Synchronization not gained\n");
            log << "Synchronization not gained\n";
            }
            else {SYNC_STAT=2;
            printf("Resynchronized\n");
            log << "Resynchronized\n";}
            }

            if (SYNC_STAT==0){
            printf("Resynchronizing at %i\n",i);
            log << "Resynchronizing at " << i << "\n";
```

```
                for (int k=0; k<16; ++k) LFSR_R[k]=bits_receiv[31-k];
                for (int k=15; k>=0; --k) R_output[31-k]=LFSR_R_fb();
                SYNC_STAT=1;

                }

                }


for (int k=1; k<47; ++k){
bits_receiv[k-1]=bits_receiv[k];
}

for (int k=1; k<32; ++k){
R_output[k-1]=R_output[k];
}
i++;

}
float BERf;
BERf= BER/counter;
printf("Errors: %i\n",BER);
log << "Errors: " << BER << "\n";
printf("Counter: %i\n",counter);
log << "Counter: " << counter << "\n";
printf("BER: %f",BERf);

return 0;
}
```